CME 250:
Introduction to Machine Learning

Lecture 8:
Neural Networks

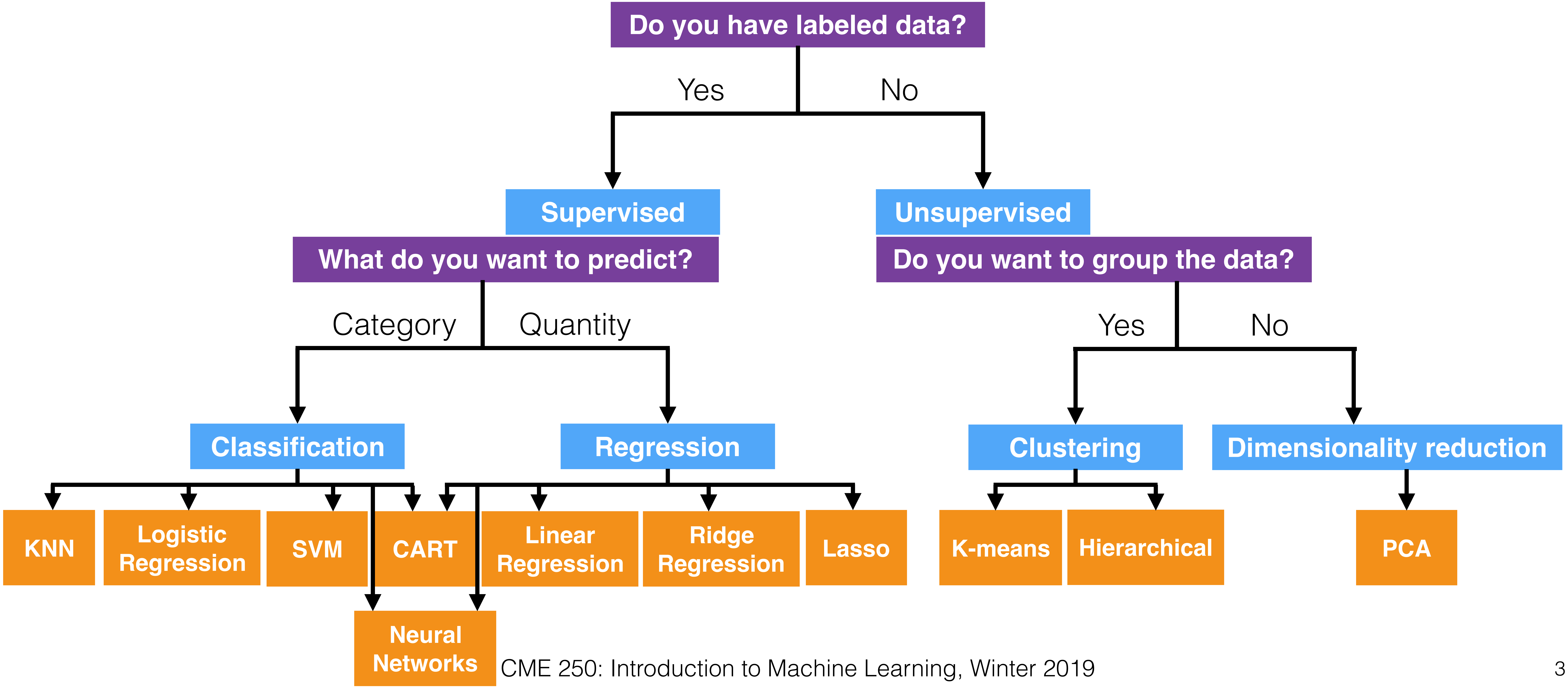
Sherrie Wang
sherwang@stanford.edu

Agenda

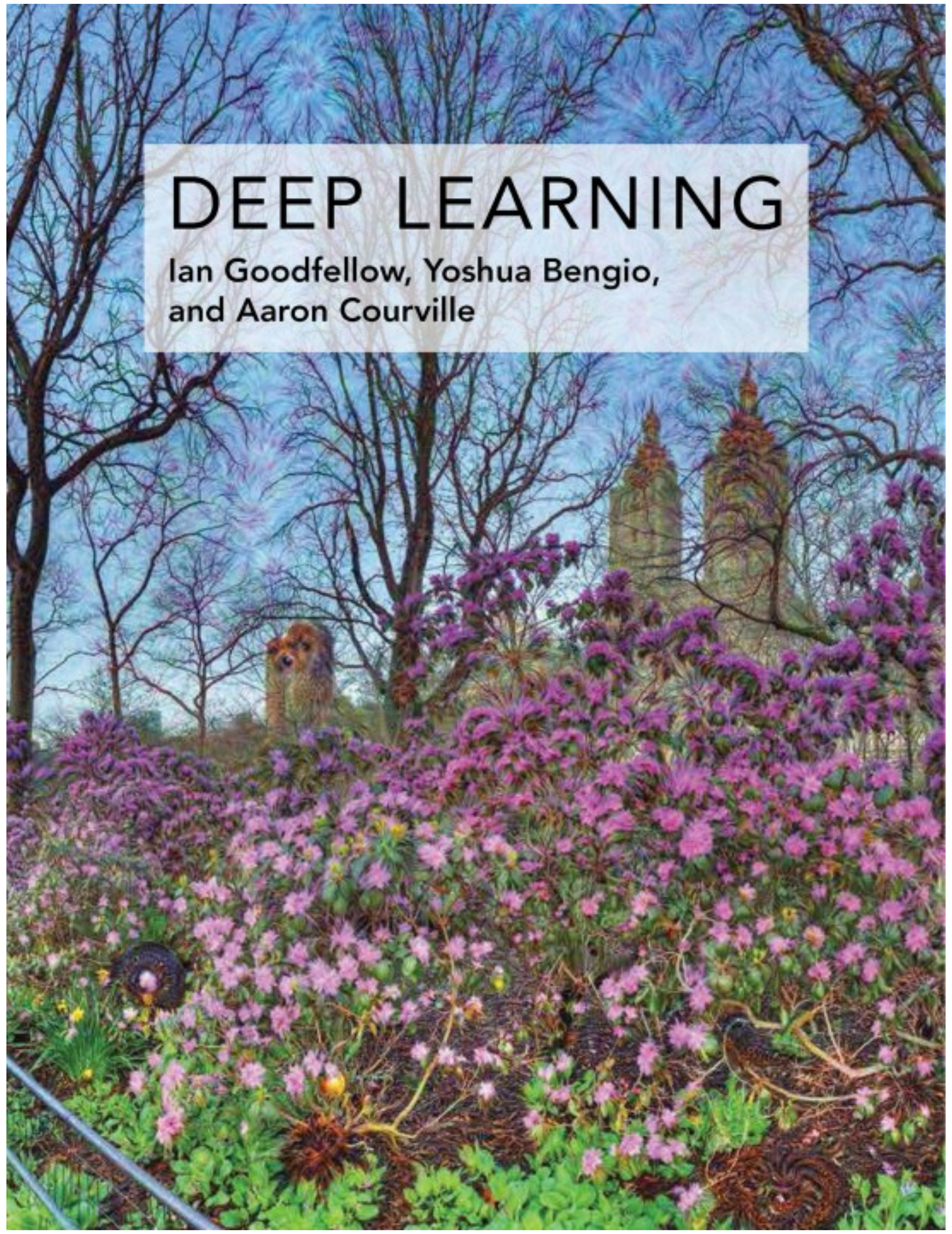
Slides are online at
cme250.stanford.edu

- Feedforward neural networks
  - Terminology and basics
  - Building blocks
  - Network architecture
  - Gradient-based learning
- Convolutional neural networks
- Recurrent neural networks

Machine Learning Methods



Deep Learning Resources



Textbook: *Deep Learning.* Ian Goodfellow, Yoshua Bengio, and Aaron Courville

Courses: CS 230, CS 231n, CS 224n

Online: *Deep learning tutorial*, *notes on CNNs*

The Success of Deep Learning

Deep Learning: A Next-Generation Big-Data Approach for Hydrology

What can Artificial Intelligence offer hydrologic research? Could deep learning one day become part of hydrology itself?

AI | By Jordan Pearson | Oct 25 2018, 9:53am

An AI-Generated Artwork Just Sold for $432,500 at Christie's

Far from being the sole creation of an AI, 'Edmond de Belamy' was the result of months of work by three people using a machine learning algorithm from 2014.

PUBLIC RELEASE: 19-JUN-2018

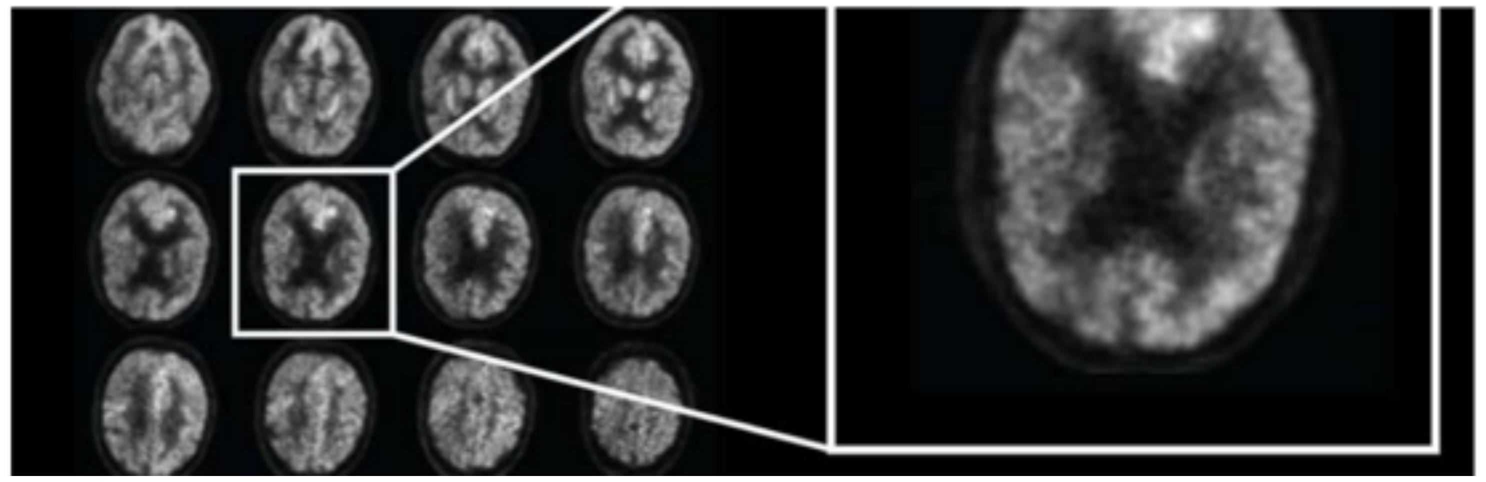
Machine learning may be a game-changer for climate prediction

COLUMBIA UNIVERSITY SCHOOL OF ENGINEERING AND APPLIED SCIENCE

SHARE PRINT E-MAIL

Artificial intelligence predicts Alzheimer's years before diagnosis

November 6, 2018, Radiological Society of North America



Machine Learning to Help Optimize Traffic and Reduce Pollution

Berkeley Lab researchers use algorithms for smart and sustainable mobility solutions

News Release Julie Chao (510) 486-6491 • OCTOBER 28, 2018

107 1 108 SHARES

14,760 views | Nov 21, 2018, 06:02pm

How Deep Learning Solves Retail Forecasting Challenges

Yuan Shen Brand Contributor
NVIDIA BRANDVOICE

Feedforward Neural Networks

Supervised Learning

Algorithms that learn to associate some input X with some output Y.

- Linear regression:

$$f(\vec{x}) = \beta_0 + \sum_{j=1}^{p} \beta_j x_j$$

- Logistic regression:

$$f(\vec{x}) = \frac{1}{1 + e^{-(\beta_0 + \sum_{j=1}^{p} \beta_j x_j)}}$$

- Support vector machine:

$$f(\vec{x}) = \beta_0 + \sum_{i \in \mathcal{S}} \alpha_i K(\vec{x}, \vec{x}^{(i)})$$

- Decision tree:

$$f(\vec{x}) = \sum_{m=1}^{M} c_m \cdot \mathbf{1}\{\vec{x} \in R_m\}$$

Linear Regression to Neural Networks

Linear models:

- Good: easy to fit, interpretable, low variance
- Bad: limited to linear functions (high bias)

SVMs:

- Use explicitly chosen kernels to model relationships beyond linear

Neural networks:

- *Learn* the kernels that best transform input to achieve output

Feedforward Neural Network

Goal: To approximate some function f^*. In the case of a model for classification or regression, want to learn $y = f^*(x)$ to map from an input x to a category or real value y. Also known as **multilayer perceptron**.

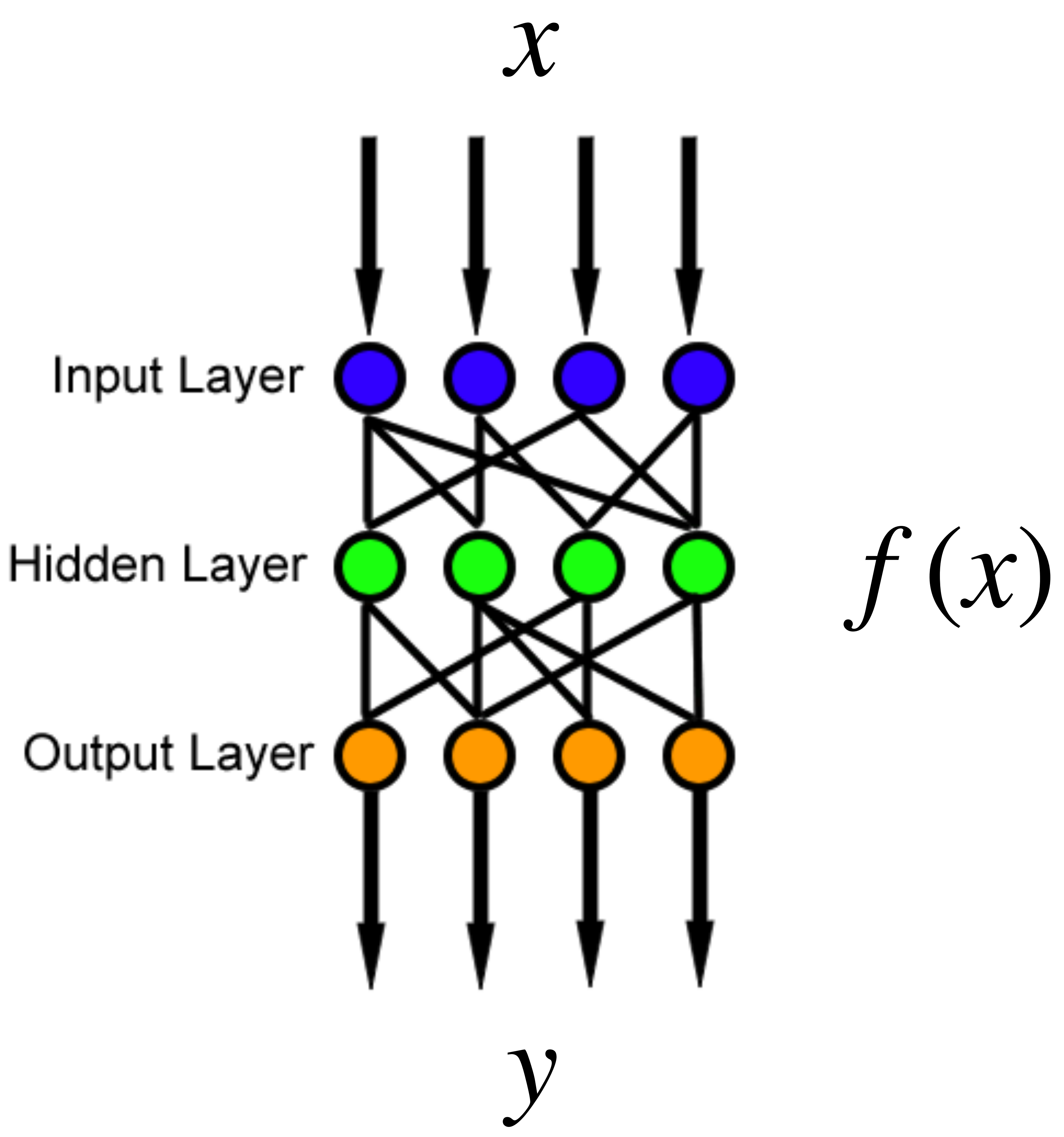
A feedforward network defines a mapping $y = f(x\ ;\ \theta)$ and learns the values of parameters θ that result in the best approximation of f^*.

Feedforward Neural Network

Feedforward: because information flows from x through the computations involved in f to the output y.

Neural: because loosely inspired by our understanding of the nervous system.

Network: because typically composes together many different functions.

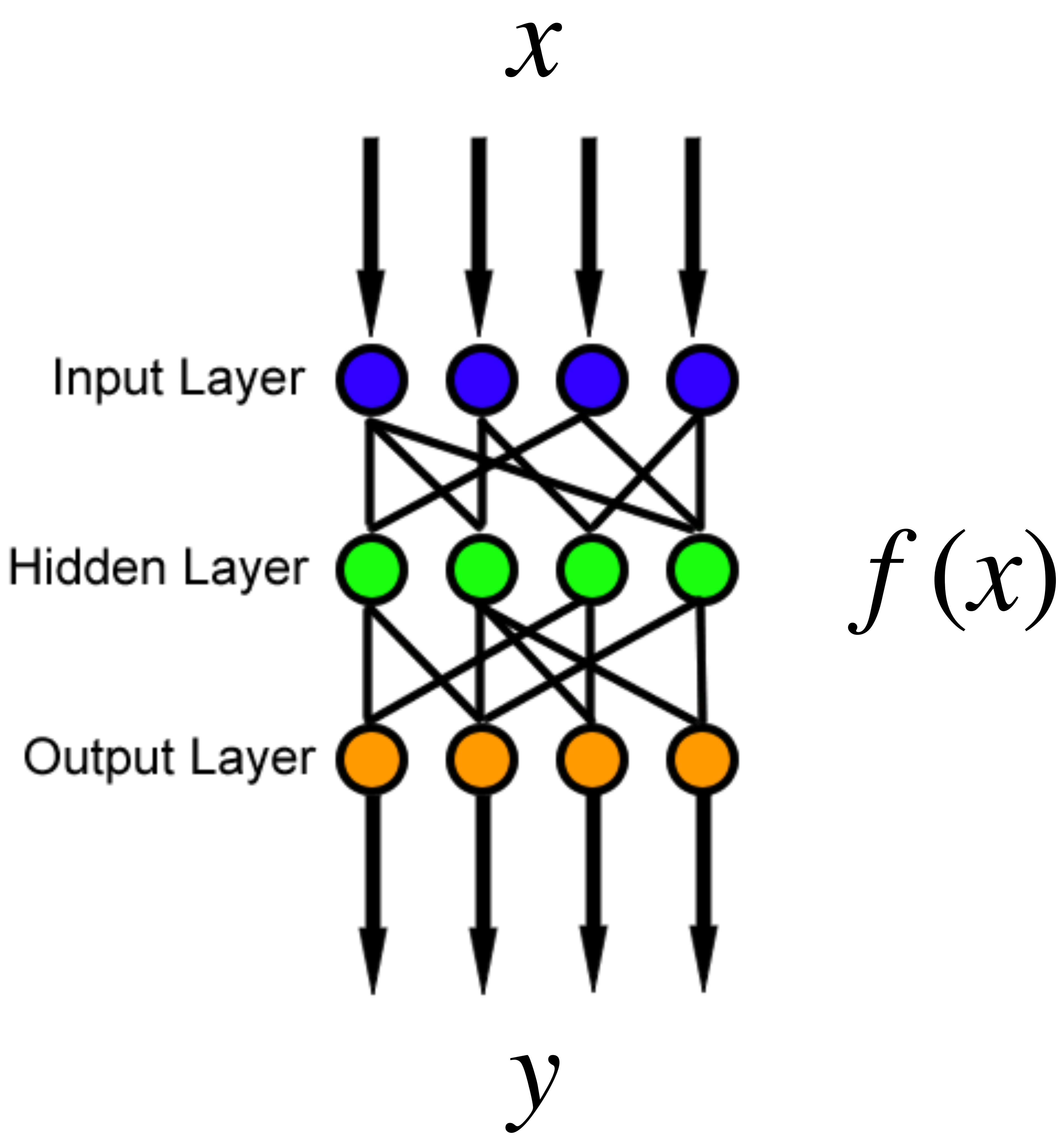


Neural Network Layers

Example: We have 3 functions $f^{(1)}$, $f^{(2)}$, and $f^{(3)}$. Connected in a chain, they form a neural network $f^{(3)}(f^{(2)}(f^{(1)}(x)))$.

In the case of the 3-layer network, $f^{(1)}$ is called the first layer, $f^{(2)}$ the second layer, and $f^{(3)}$ the third layer.

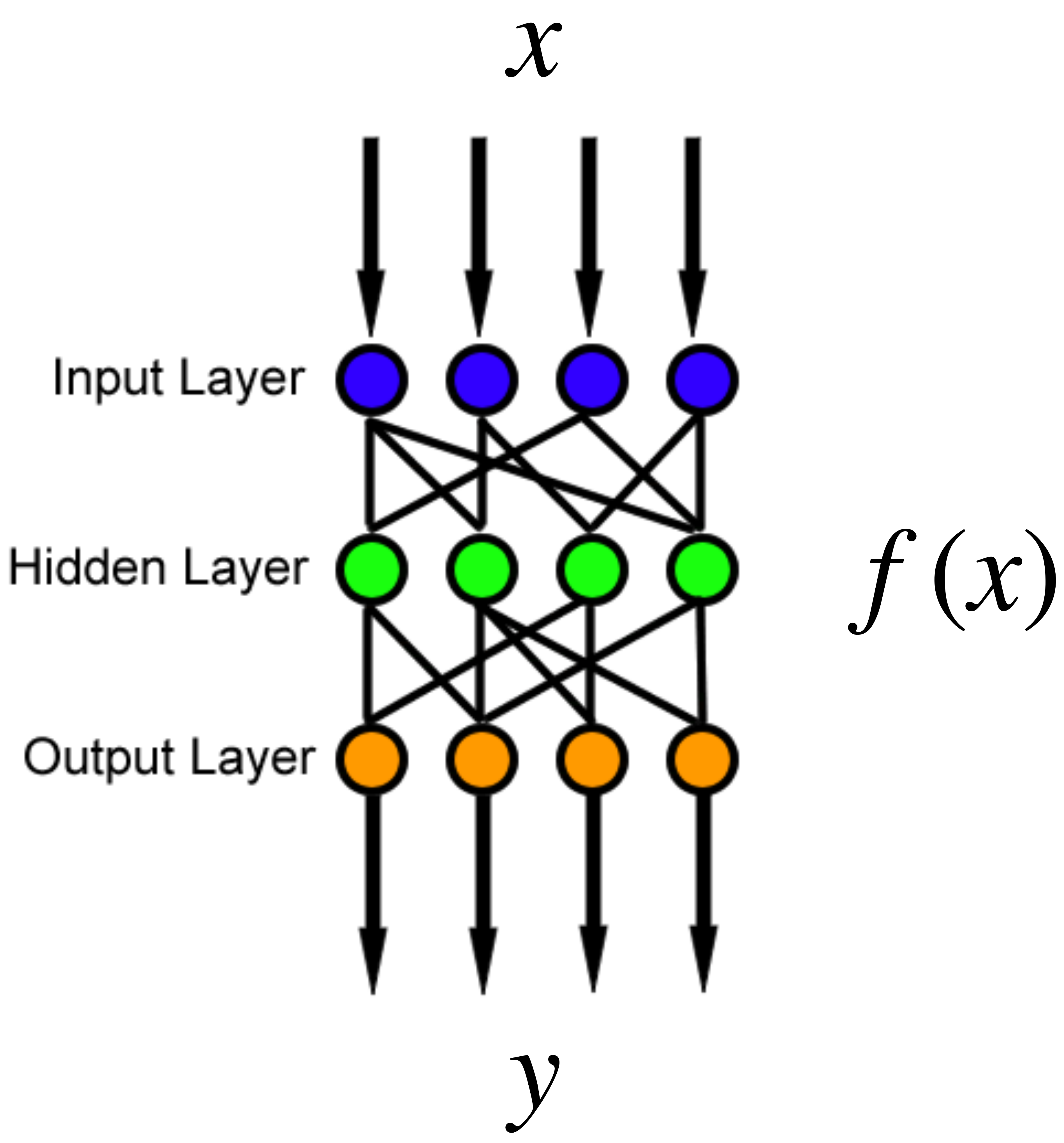
The number of layers is the **depth** of the model.



Neural Network Layers

When we train a neural network, we want to drive f to be close to f^*.

Of course, we don't know f^*; we just have training data $(x^{(i)}, y^{(i)})$. For each $x^{(i)}$, we want the value from the **output layer** of the network to match $y^{(i)}$.



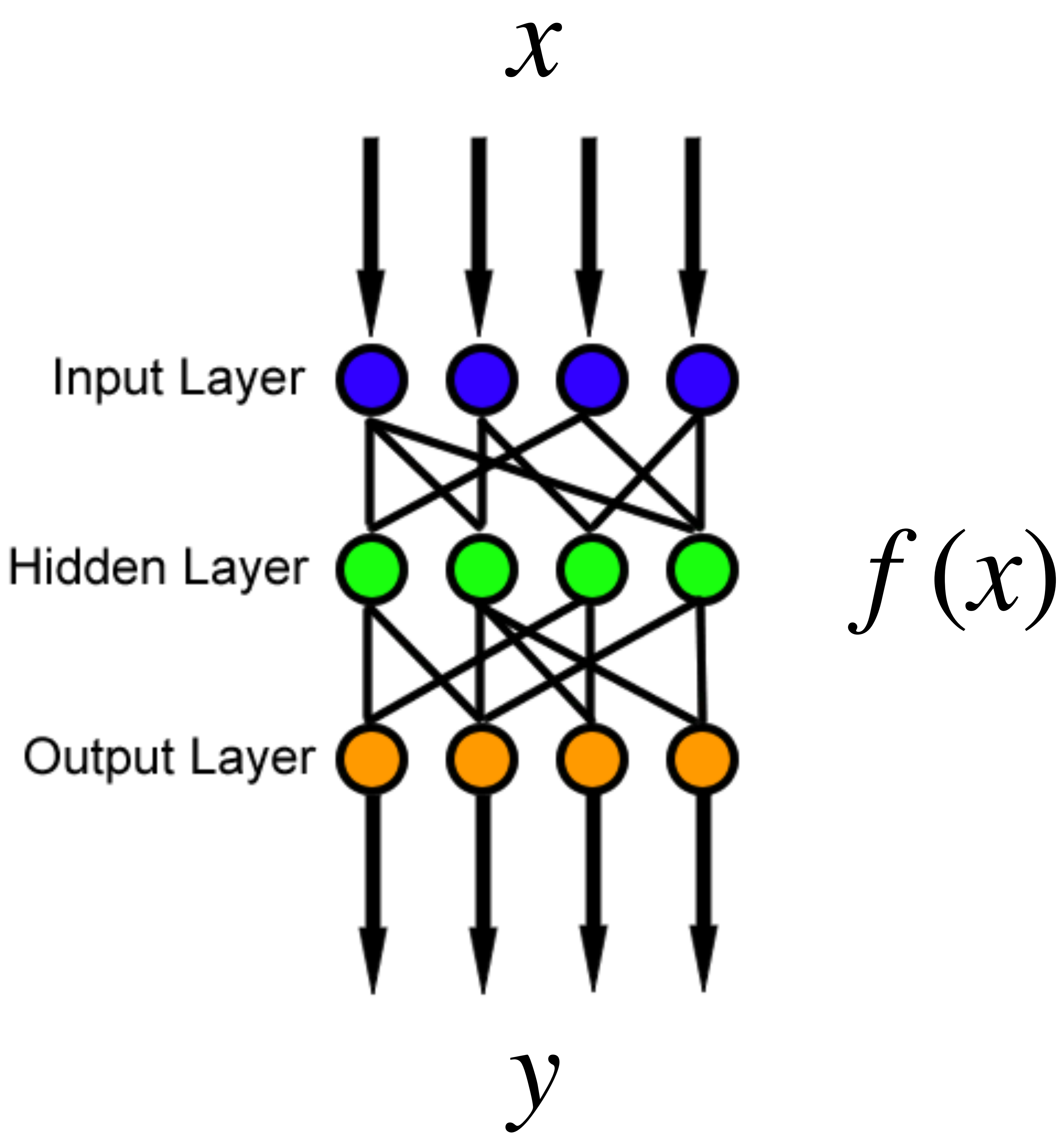
Neural Network Layers

Behavior of intermediate layers is not directly specified by the training data, so we call these layers **hidden layers**.

Representing hidden layers as vectors, maximal dimension = **width** of model.

Each element of hidden layer is a **unit**.

Functions used to compute hidden layer values are called **activation functions**.

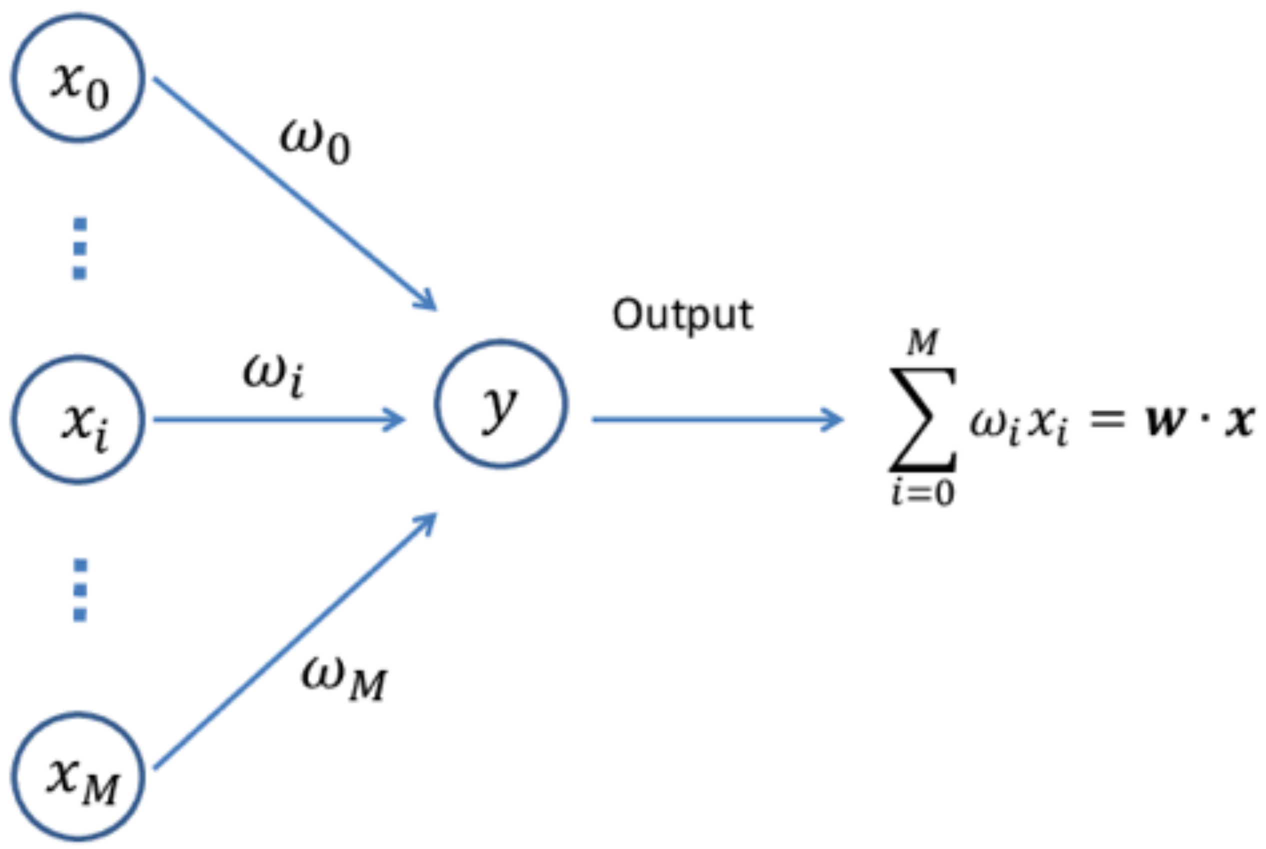


Neural Networks: Examples

A linear 1-layer neural network:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2$$

$$y = \mathbf{X}\beta$$



A nonlinear 3-layer neural network:

$$y = f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{X}\mathbf{W}_1)\mathbf{W}_2)\mathbf{W}_3)$$

https://www.researchgate.net/publication/316613684_Heterogeneous_sharpness_for_cross-spectral_face_recognition/figures?lo=1

Universal Approximation Theorem

It might seem that in order to approximate arbitrary nonlinear functions, we have to choose the right model family for that function.

The **universal approximation theorem** (Hornik *et al.* 1989; Cybenko, 1989) states that a feedforward network with a linear output layer and at least 1 hidden layer with any “squashing” activation function (e.g. sigmoid function) can approximate any function from one finite-dimensional space to another with any nonzero amount of error, provided the network has enough hidden units.

In the worst case, $O(2^n)$ hidden units are needed.

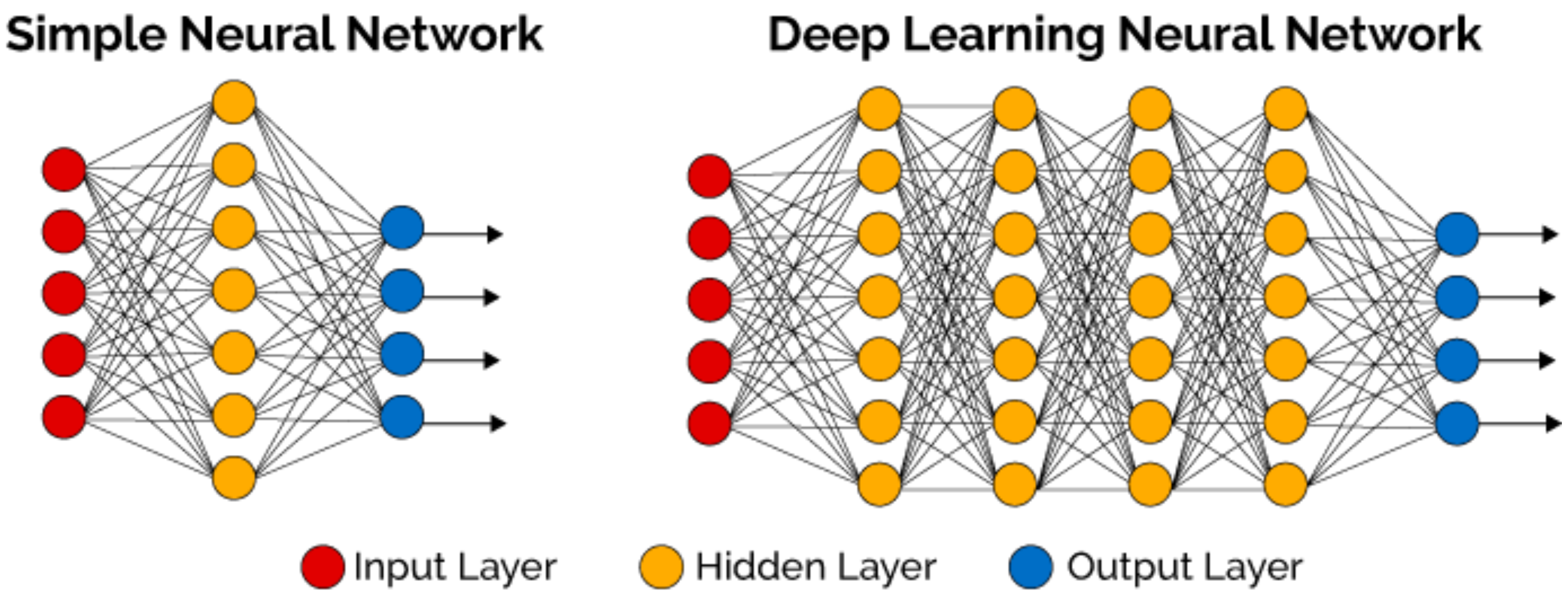
Deep Learning

$O(2^n)$ hidden units is not computationally feasible.

Instead of making model wider (results guaranteed eventually by universal approximation theorem), make the model deeper.

In practice, compositions of simple nonlinear functions can approximate complex nonlinear functions.

Deep Learning



README.md

Deep Residual Networks with 1K Layers

By Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun.

Microsoft Research Asia (MSRA).

Building Blocks of Neural Networks

- Linear transformations
- Nonlinear transformations
  - Activation functions
- Obtaining outputs
  - Output functions

Linear Transformations

Some main building blocks of feedforward neural networks is shared with linear regression: addition and multiplication.

The **weights** of a neural network are real-valued matrices multiplied by the inputs to each layer. They are learned via training.

$$y = f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{X}\mathbf{W}_1)\mathbf{W}_2)\mathbf{W}_3)$$

Nonlinear Transformations

The activation functions of hidden layers are simple nonlinear functions. They are determined when the network architecture is coded and do not change during training.

$$y = f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{X}\mathbf{W}_1)\mathbf{W}_2)\mathbf{W}_3)$$

Obtaining Outputs

The output layer is usually just a linear transformation for regression problems and a linear transformation followed by some “squashing” function that brings a real value into the interval $(0,1)$ for classification problems.

$$y = f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{X}\mathbf{W}_1)\mathbf{W}_2)\mathbf{W}_3)$$

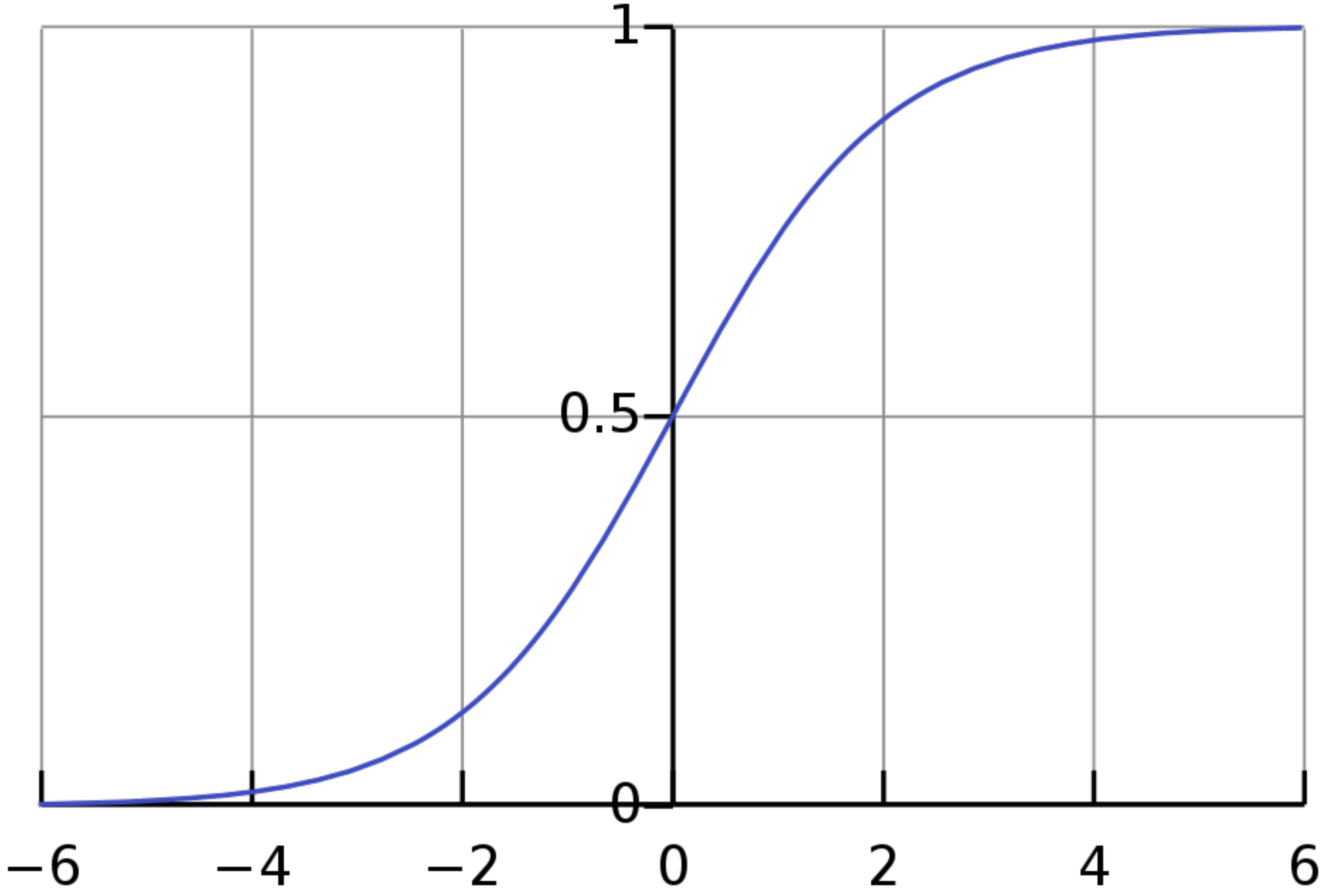
Activation Functions

Which activation functions are best and the theoretical principles guiding their design are still an active area of research.

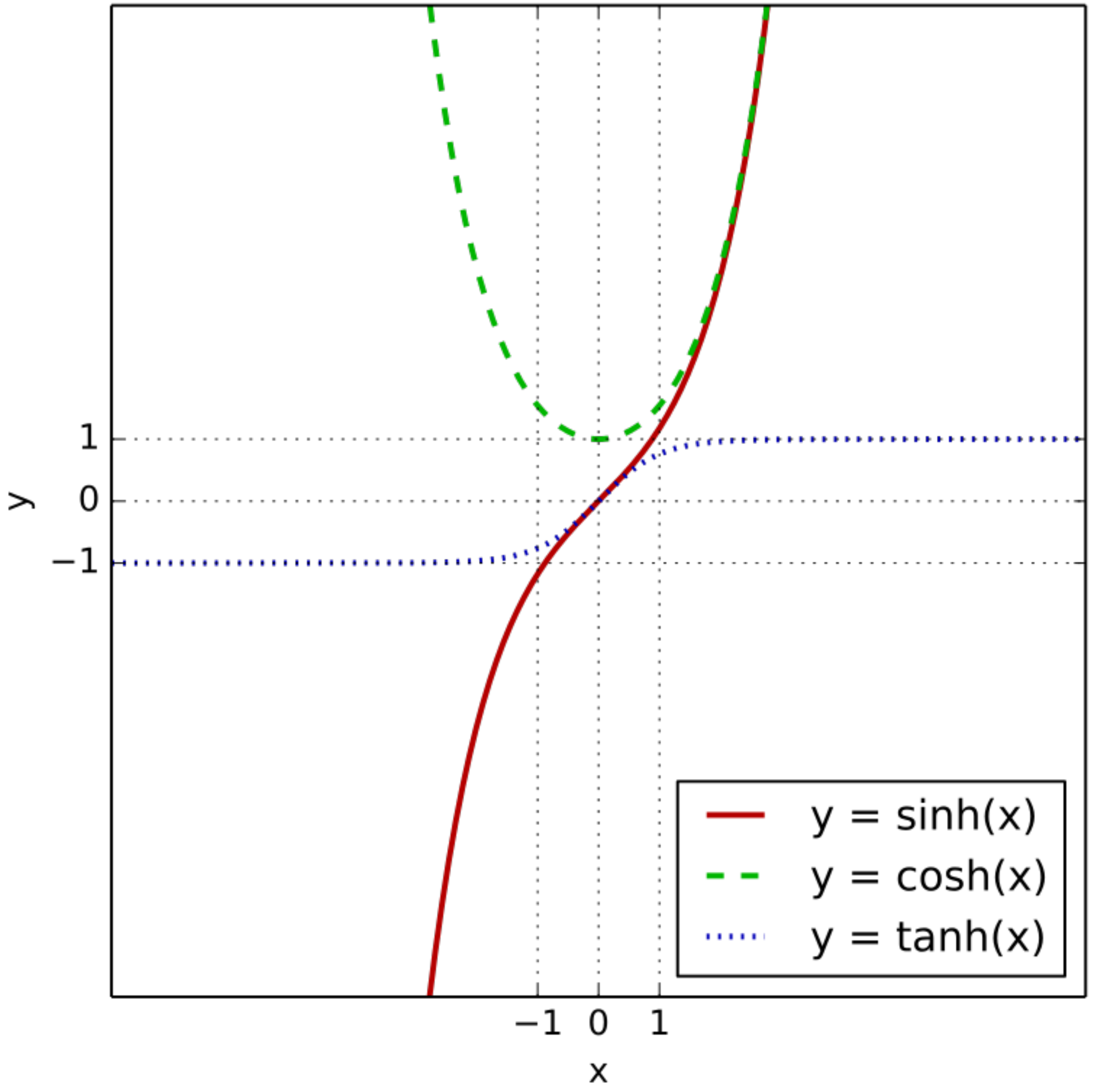
Common activation functions include:

- **Logistic sigmoid**
- **Hyperbolic tangent**
- **Rectified linear units**

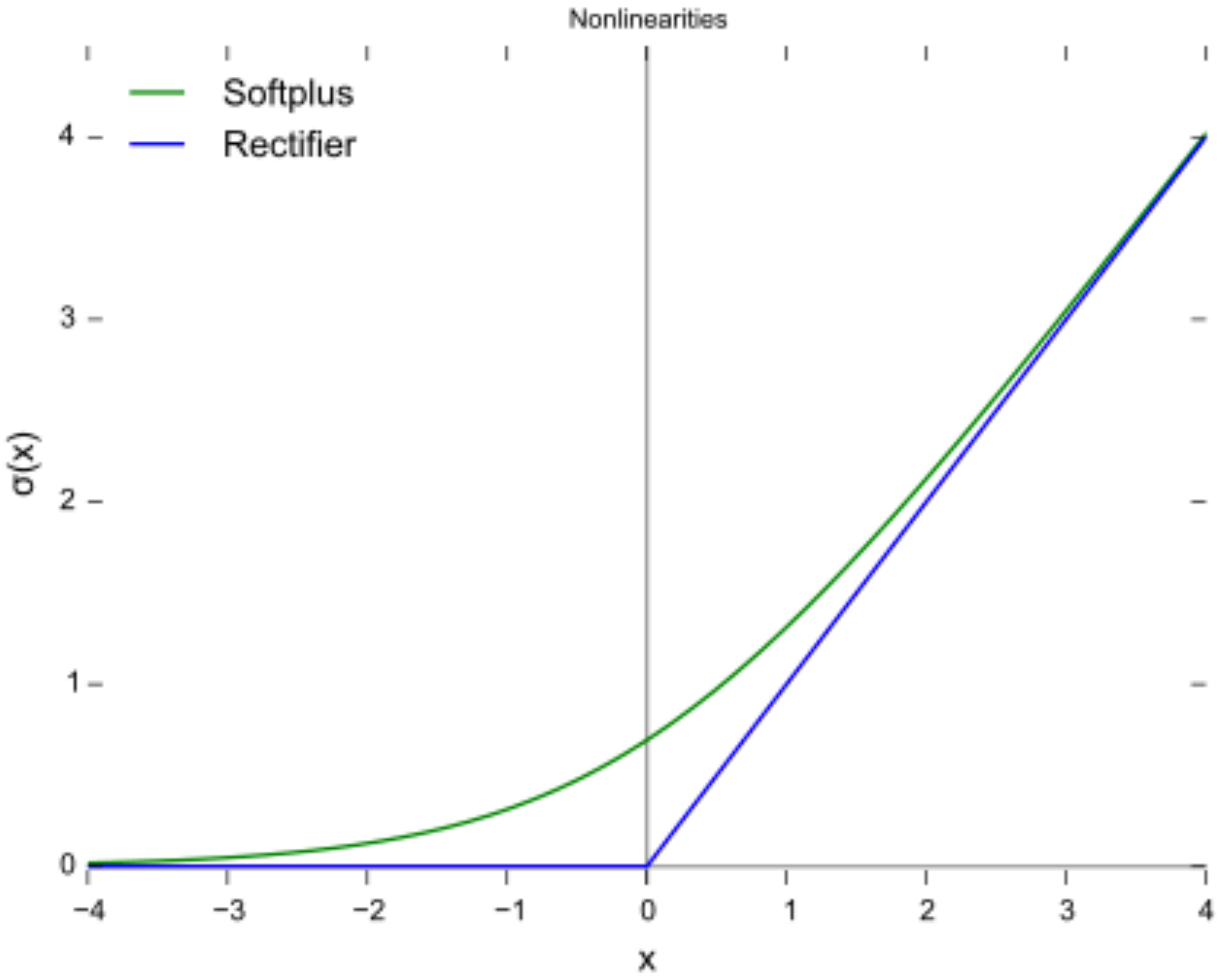
Sigmoid Function



Hyperbolic Tangent Function



Rectified Linear Unit (ReLU)



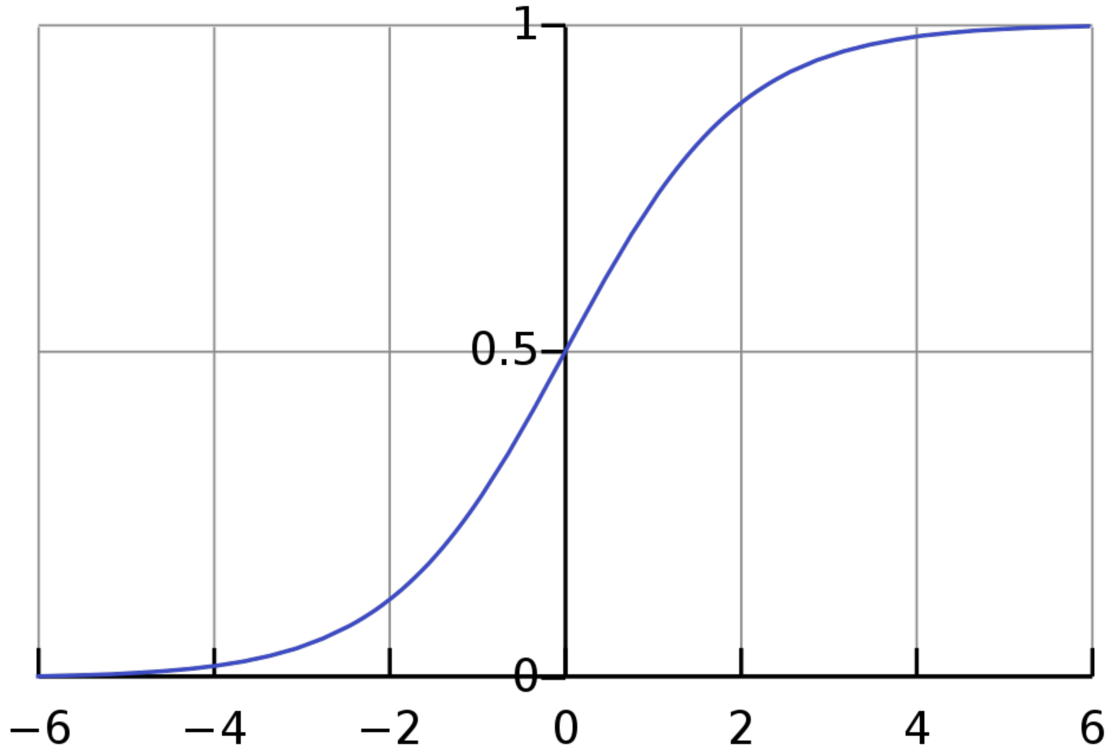
Output Functions

For real-valued outputs, use a **linear** output function.

$$\hat{y} = \mathbf{W}^\top \mathbf{h} + \mathbf{b}$$

For binary outputs, use a **sigmoid** output function.

$$\hat{y} = \sigma(\mathbf{W}^\top \mathbf{h} + \mathbf{b})$$



For multi-class outputs, use a **softmax** output function.

$$\hat{y}_k = \frac{e^{(\mathbf{W}^\top \mathbf{h}+\mathbf{b})_k}}{\sum_j e^{(\mathbf{W}^\top \mathbf{h}+\mathbf{b})_j}}$$

Each element is in $(0,1)$.
Entire vector sums to 1.

Network Architecture

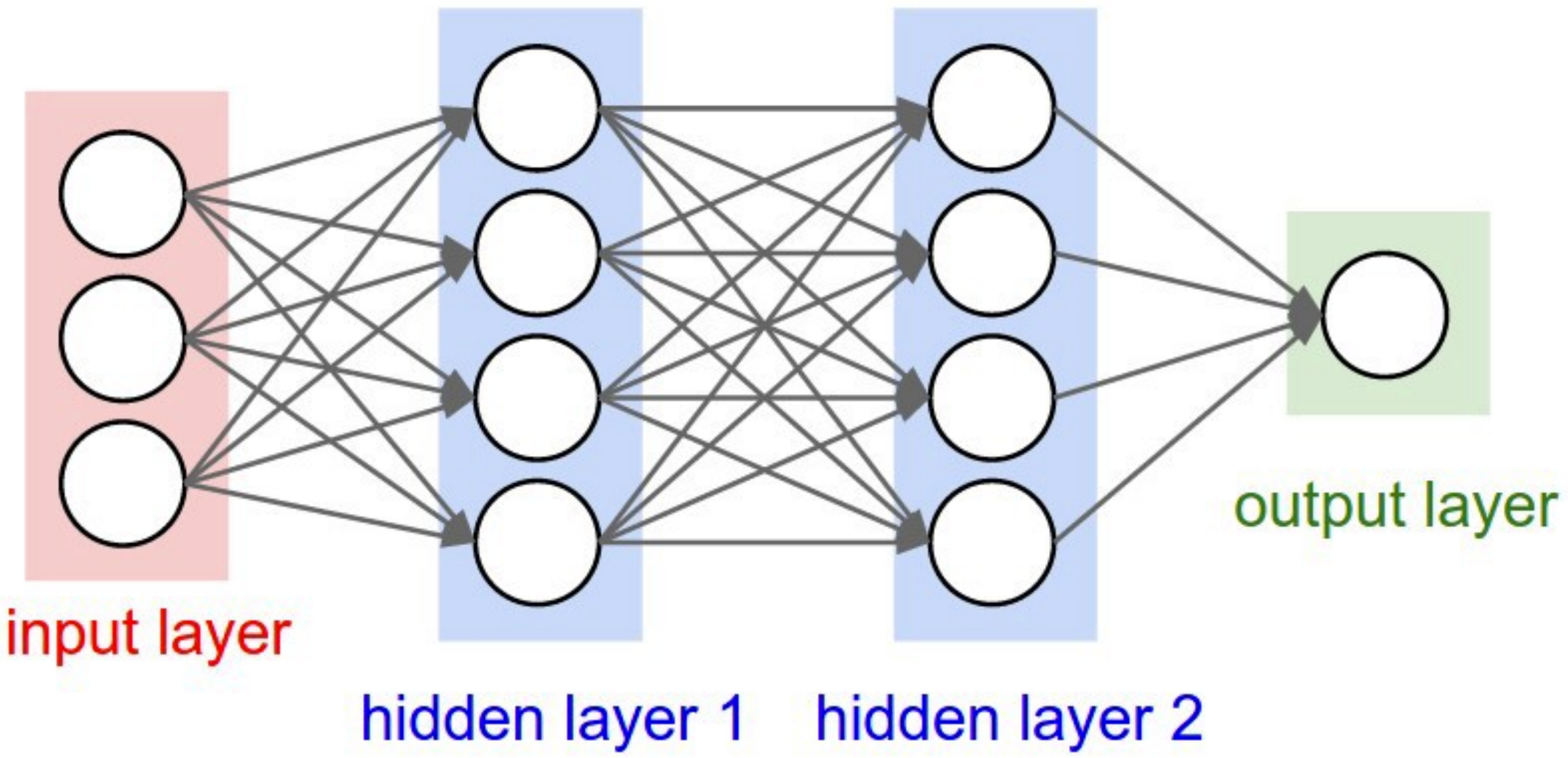
Architecture: the overall structure of the network. How many units it has, how these units are connected to each other.

Example:

$$\mathbf{h}^{(1)} = \mathrm{ReLU}(\mathbf{W}^{(1)\top}\mathbf{x} + \mathbf{b}^{(1)})$$

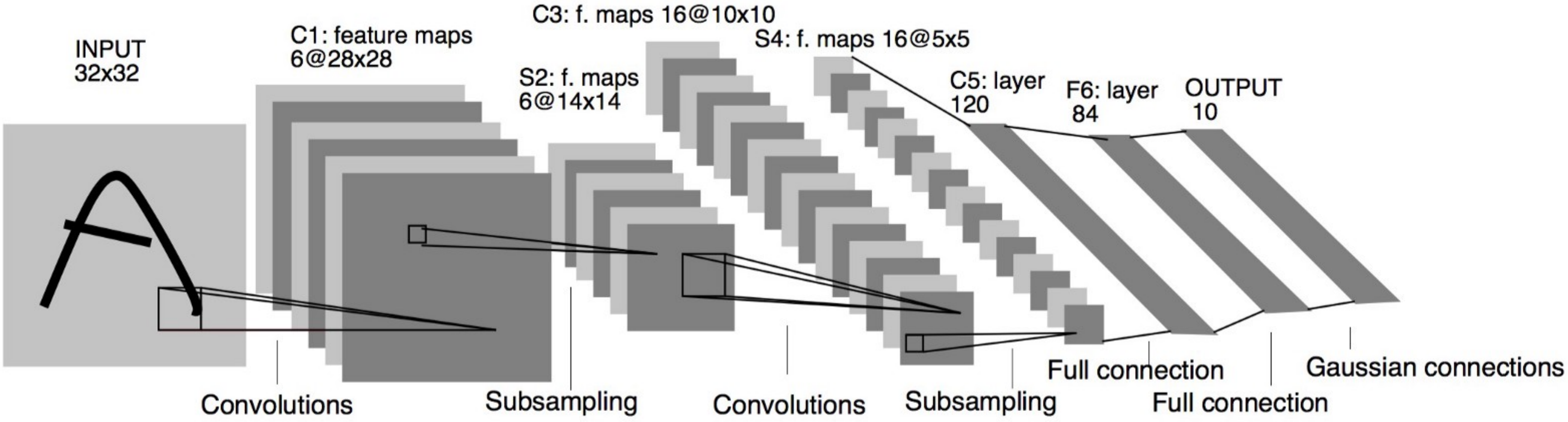
$$\mathbf{h}^{(2)} = \mathrm{ReLU}(\mathbf{W}^{(2)\top}\mathbf{h}^{(1)} + \mathbf{b}^{(2)})$$

$$\hat{y} = \sigma(\mathbf{W}^{(3)\top}\mathbf{h}^{(2)} + \mathbf{b}^{(3)})$$



Network Architecture

Nowadays, there are lots of network architectures to choose from. Try existing ones before customizing for your own application.



Gradient-based Learning

How do we actually learn the network weights W?

Linear regression has a closed-form solution. It is also convex and can be solved via convex optimization.

SVMs are convex and can be solved via convex optimization.

Decision trees are built via greedy algorithm.

Optimizing neural networks is a non-convex problem.

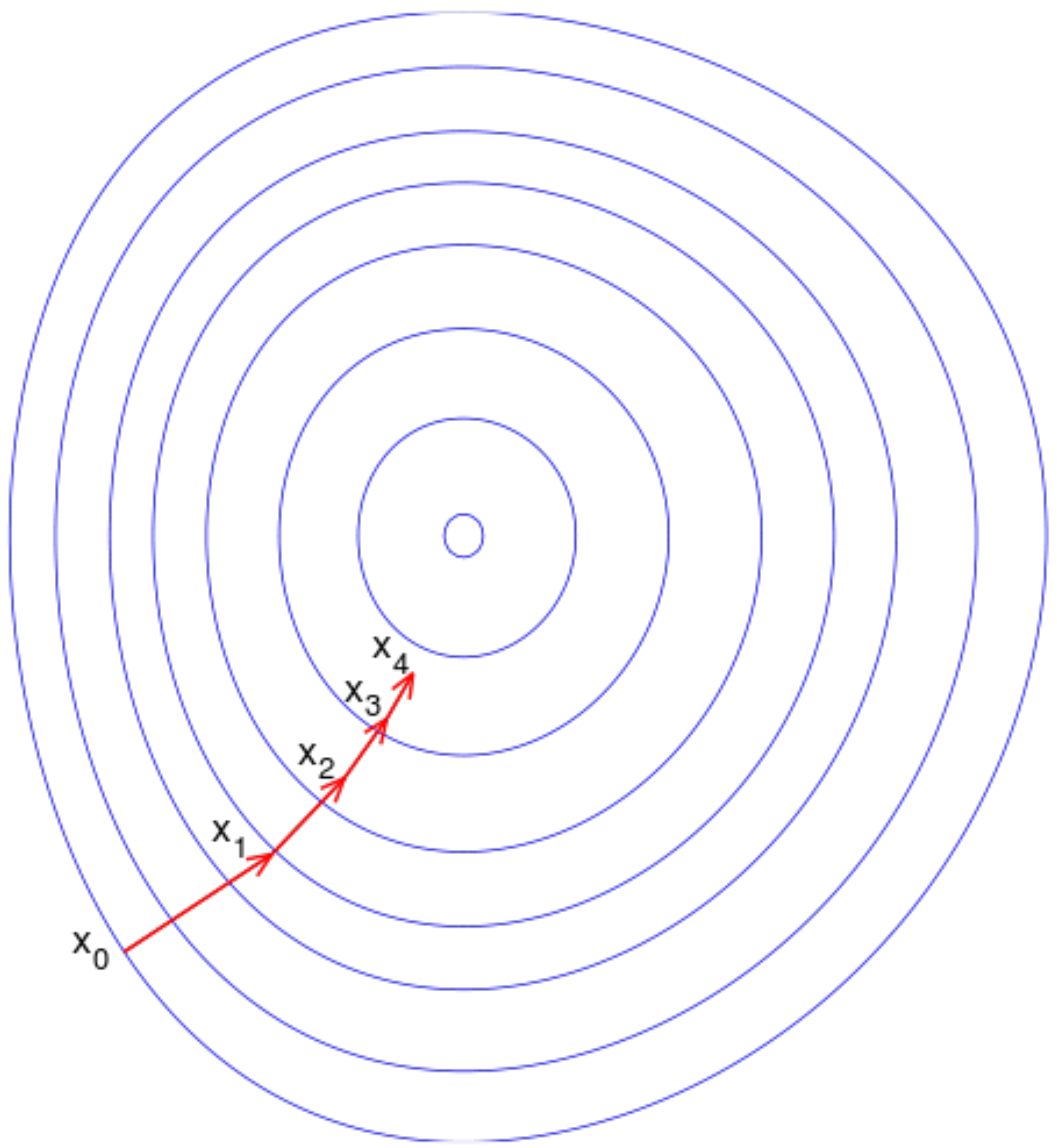
Gradient Descent

An iterative optimization algorithm for finding the minimum of a function $F(x)$.

Move in the direction in which $F(x)$ decreases the most.

Guaranteed to find global minimum of convex functions, but not non-convex functions.

In practice, local minima of neural network weights are often pretty good.

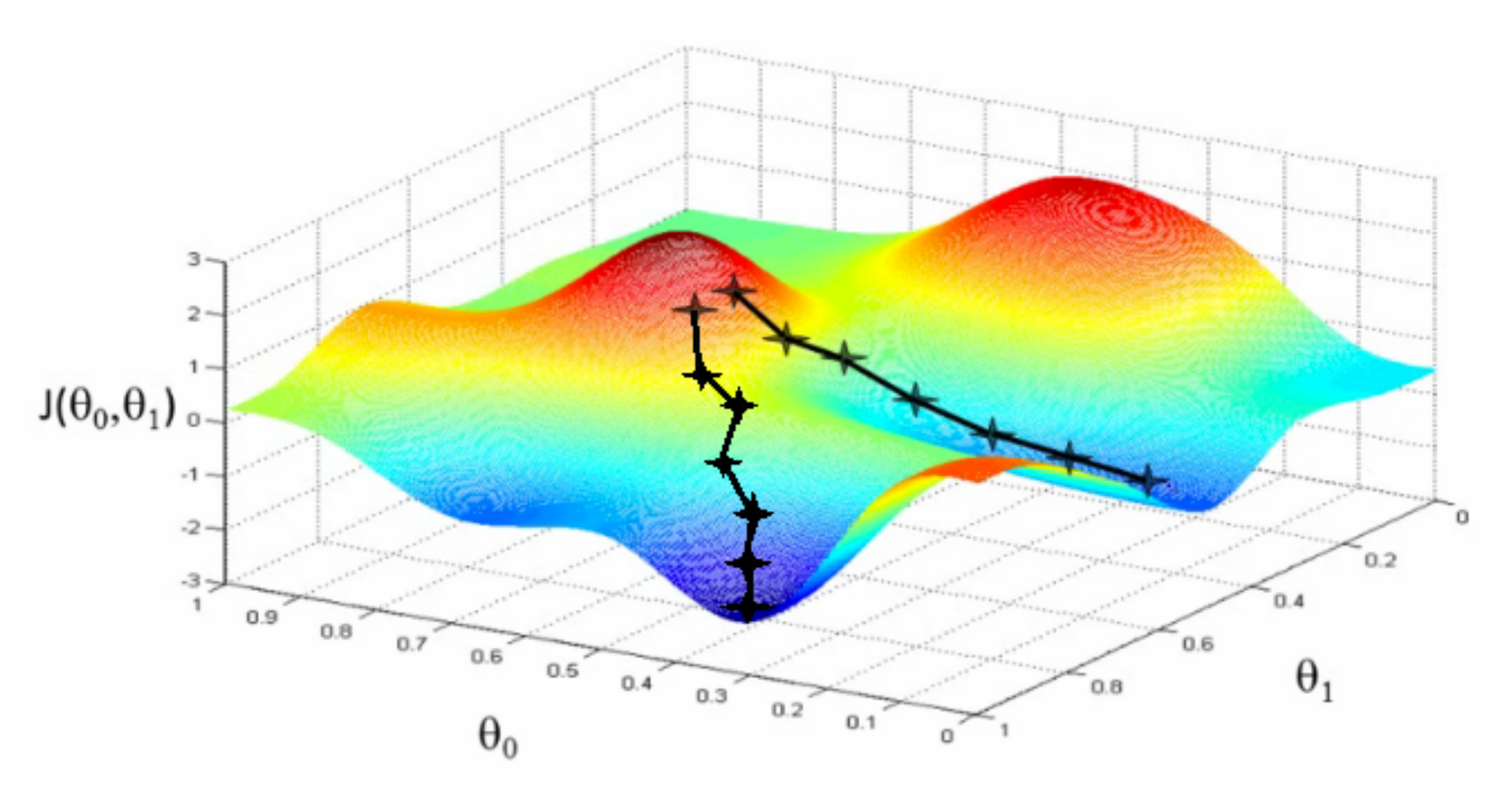


Cost Functions

On what function do we run gradient descent? **Cost function**.

Intuitively, a cost function measures how well we are doing in our task of interest.

Examples: Regression might use mean squared error, classification the maximum likelihood of observing the training data.



Cost Function: Example

Suppose the cost function is MSE:

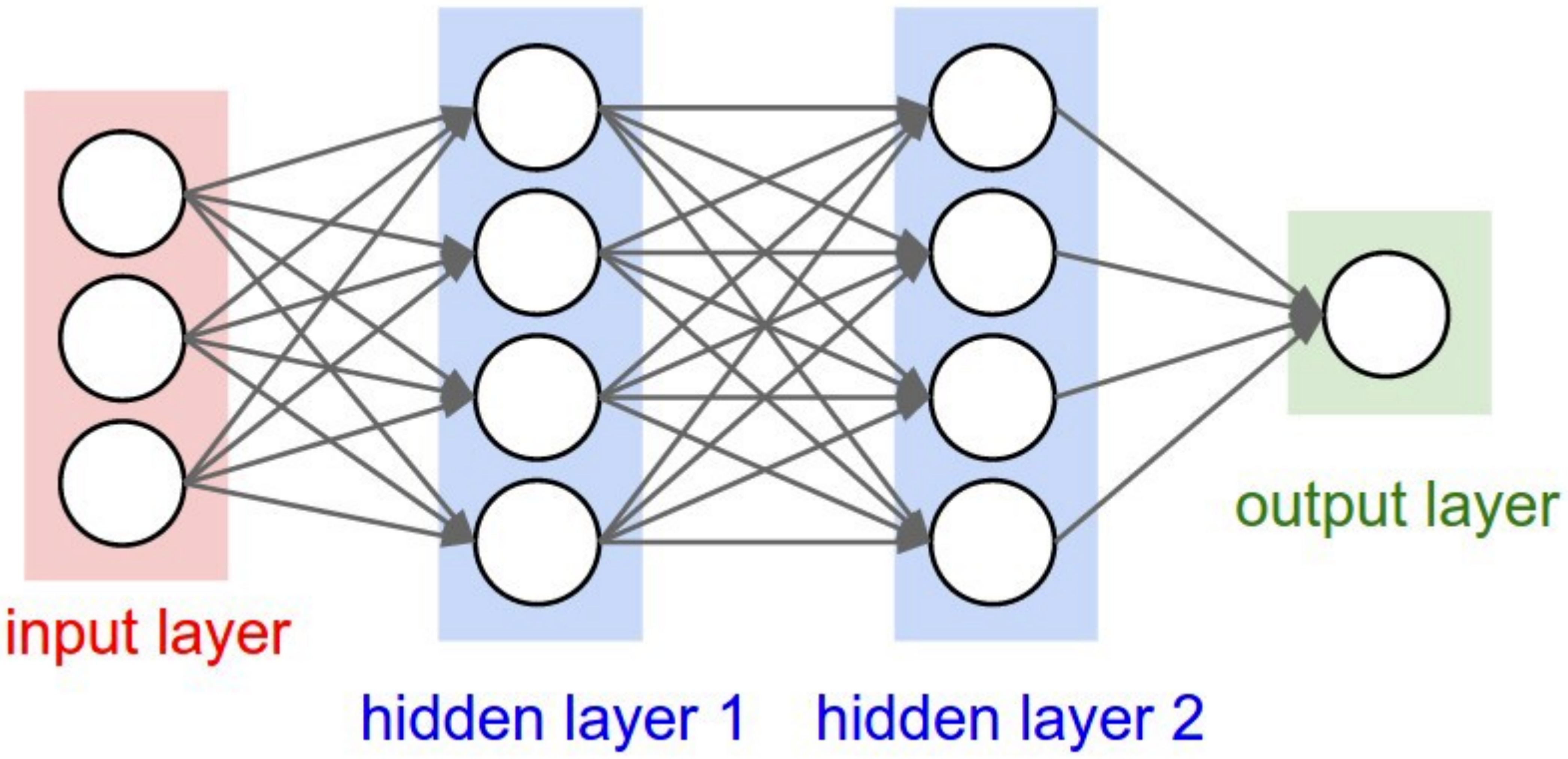
Then for the network at right:

$$\mathbf{h}^{(1)} = \mathrm{ReLU}(\mathbf{W}^{(1)\top}\mathbf{x} + \mathbf{b}^{(1)})$$

$$\mathbf{h}^{(2)} = \mathrm{ReLU}(\mathbf{W}^{(2)\top}\mathbf{h}^{(1)} + \mathbf{b}^{(2)})$$

$$\hat{y} = \mathbf{W}^{(3)\top}\mathbf{h}^{(2)} + \mathbf{b}^{(3)}$$

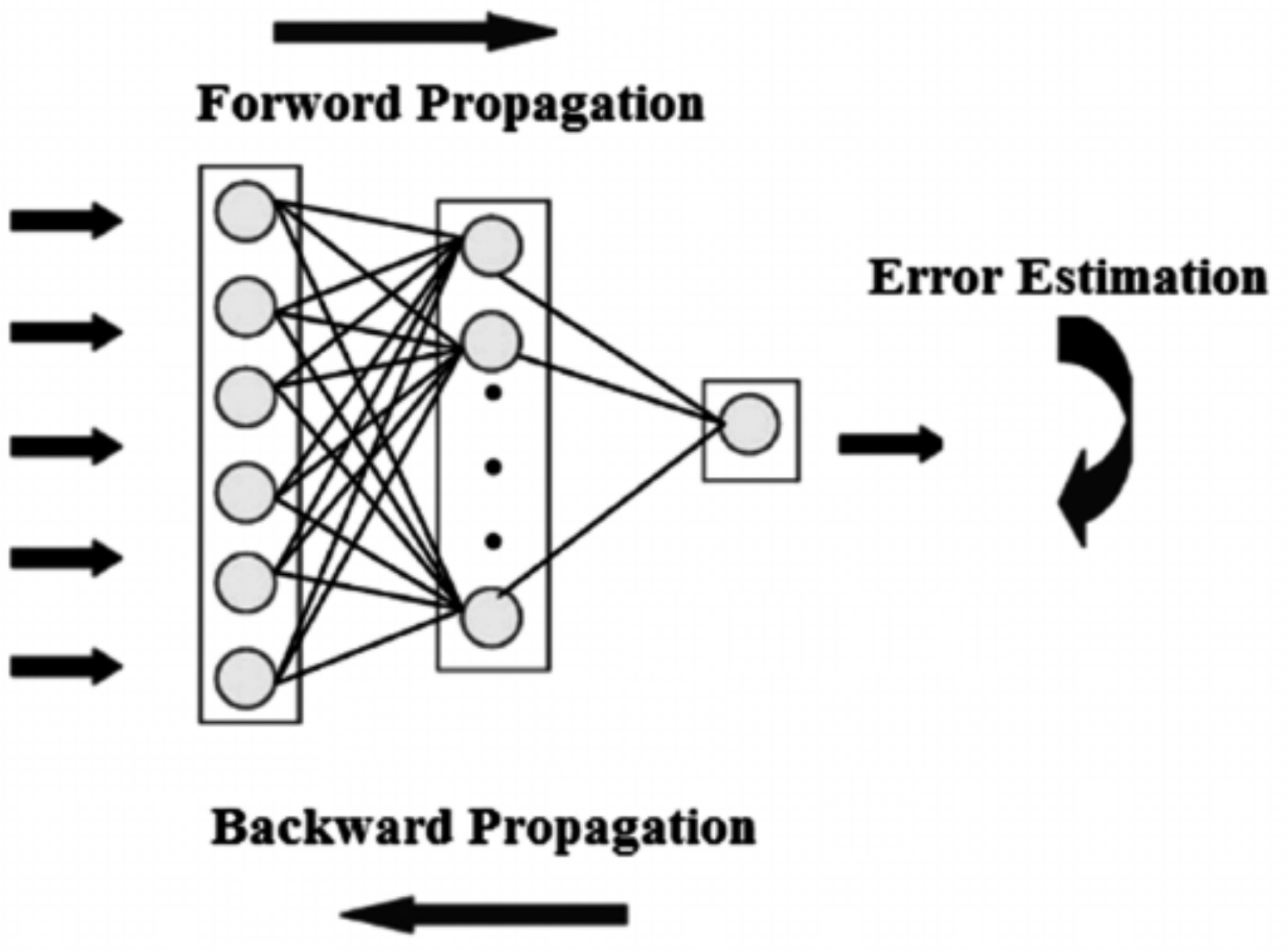
$$J(\mathbf{W}, \mathbf{b}) = \frac{1}{n}\sum_{i=1}^{n} ||y^{(i)} - f(\mathbf{x}^{(i)}; \mathbf{W}, \mathbf{b})||^2$$



Backward Propagation

Find the direction of maximum decrease (negative gradient) of the cost function w.r.t. the weights, and move the weights in this direction.

The process of finding the gradient of the cost function w.r.t. the weights is called **backward propagation**, or backprop.



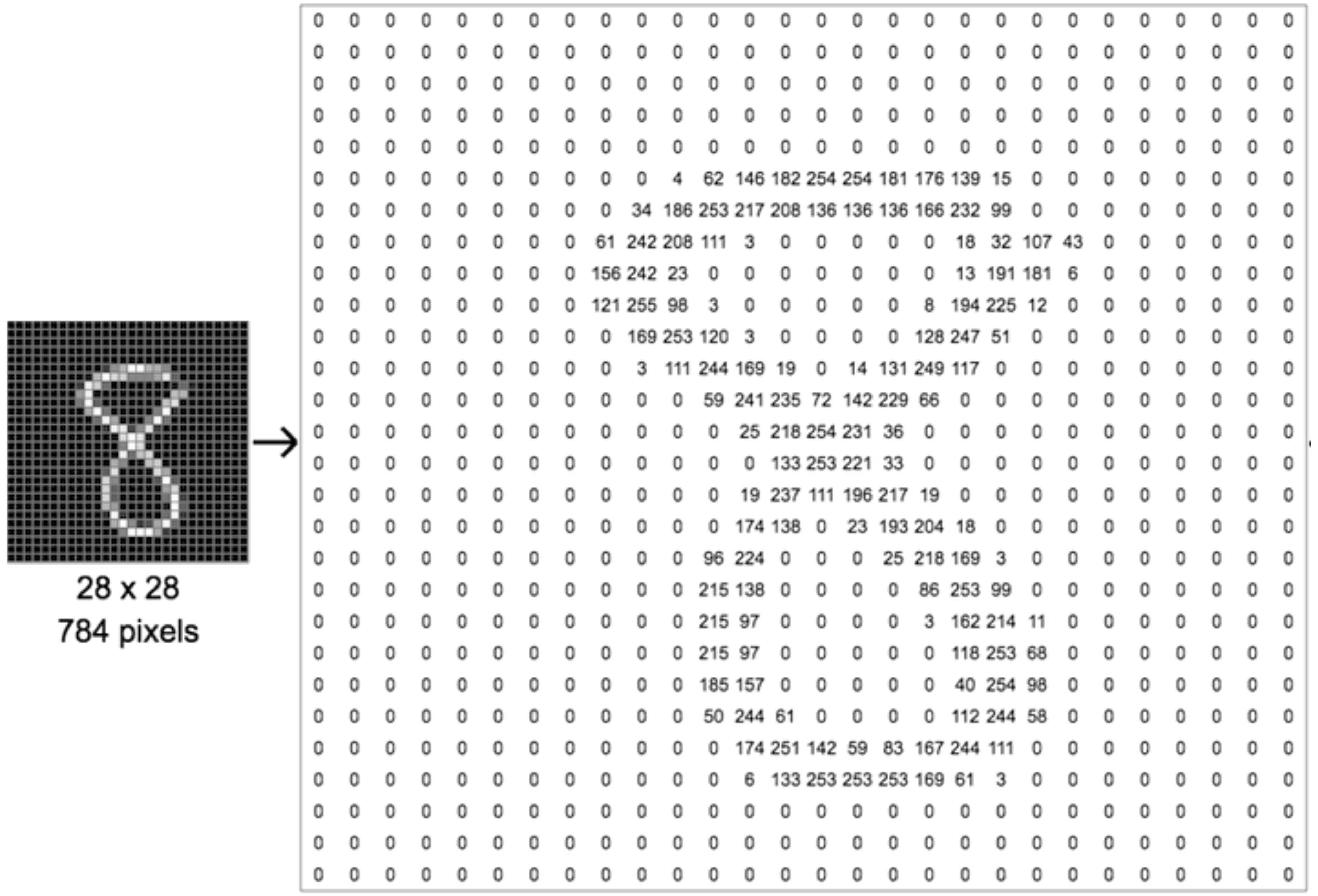
Convolutional Neural Networks (CNN)

Images as Inputs

An image can be represented as a matrix of integers or real values.

If input to a traditional feedforward network, it can be flattened into a vector and a weight matrix W multiplied with it to produce the first hidden layer.

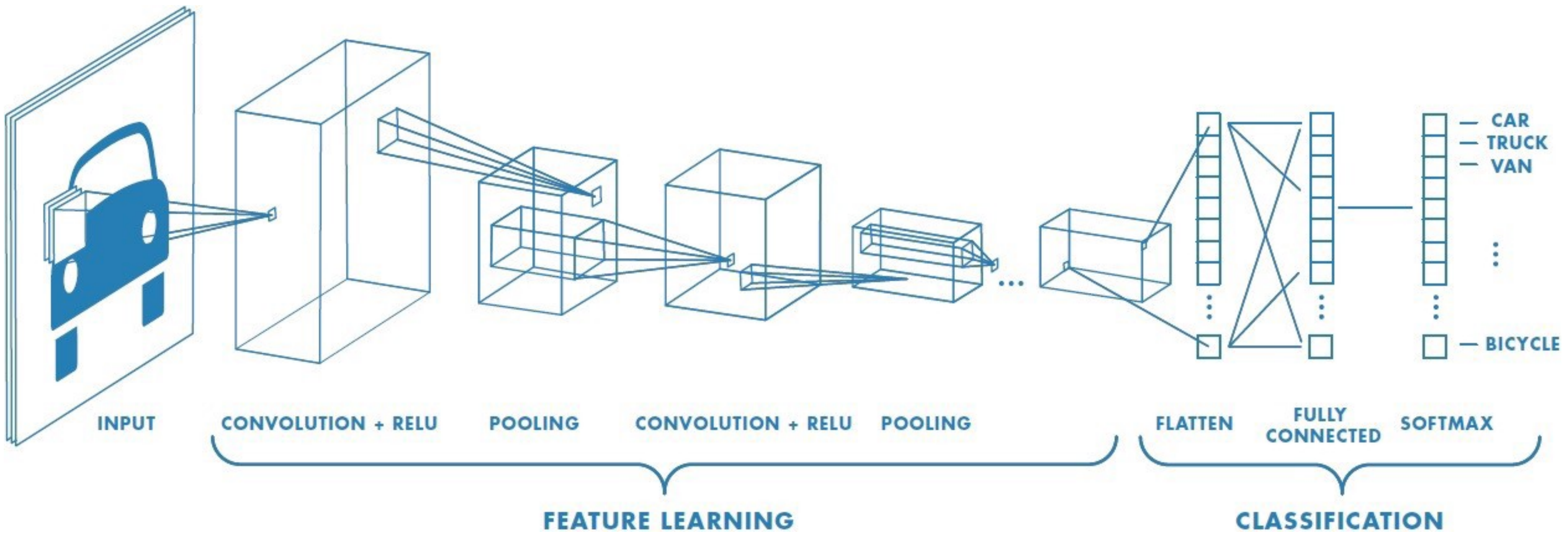
Images are high dimensional so this results in a very wide network.



Convolutional Neural Networks

A class of deep neural networks with an architecture designed to be invariant to shifts in the input. Most commonly used in image tasks.

New layer types: Convolutional layer, Pooling layer



Convolutional Layers

A convolutional layer is comprised of **filters**, which are small matrices.

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

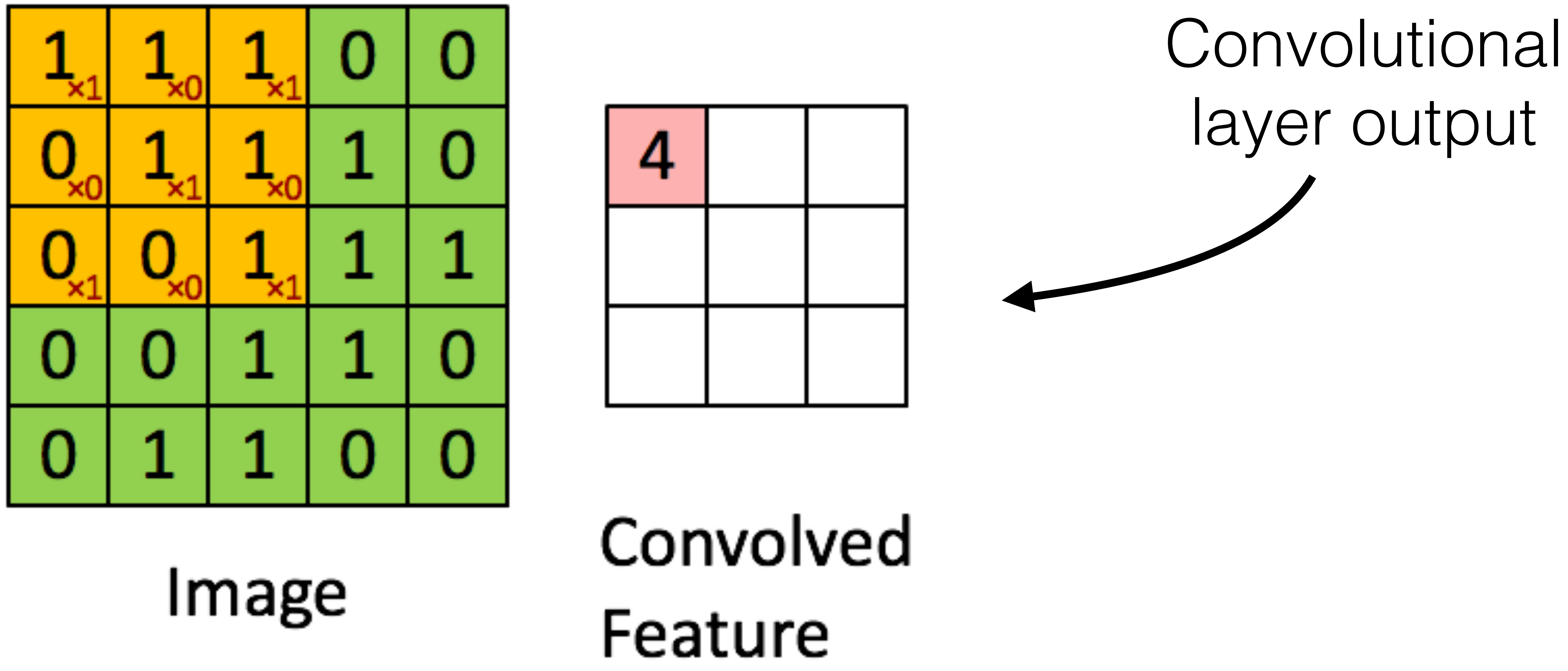
Input

1	0	1
0	1	0
1	0	1

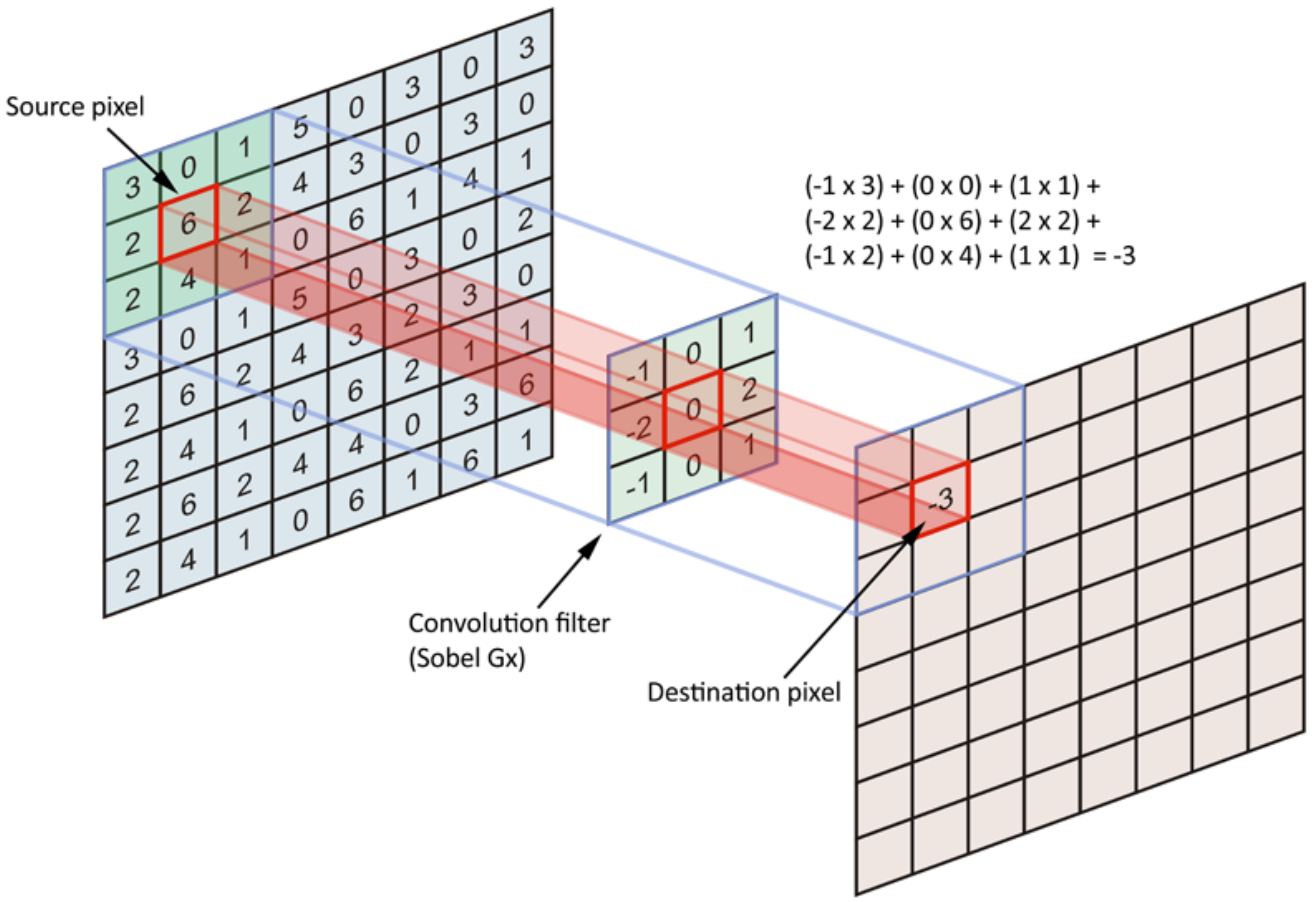
Filter / Kernel

Convolutional Layers

Instead of taking the product between the entire image and the weights, a filter convolves with each filter-sized piece of the image.

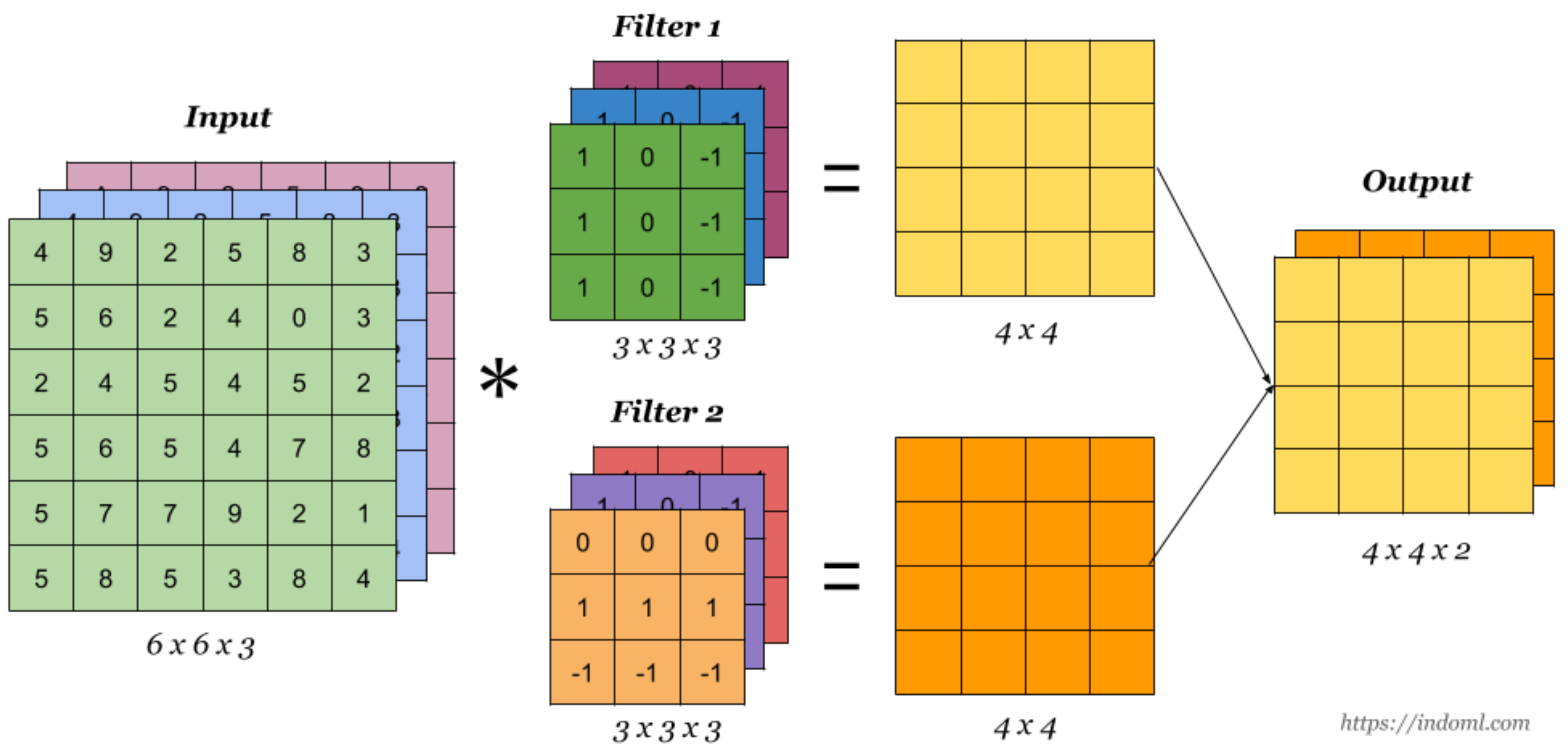


Convolutional Layers



Convolutional layer output

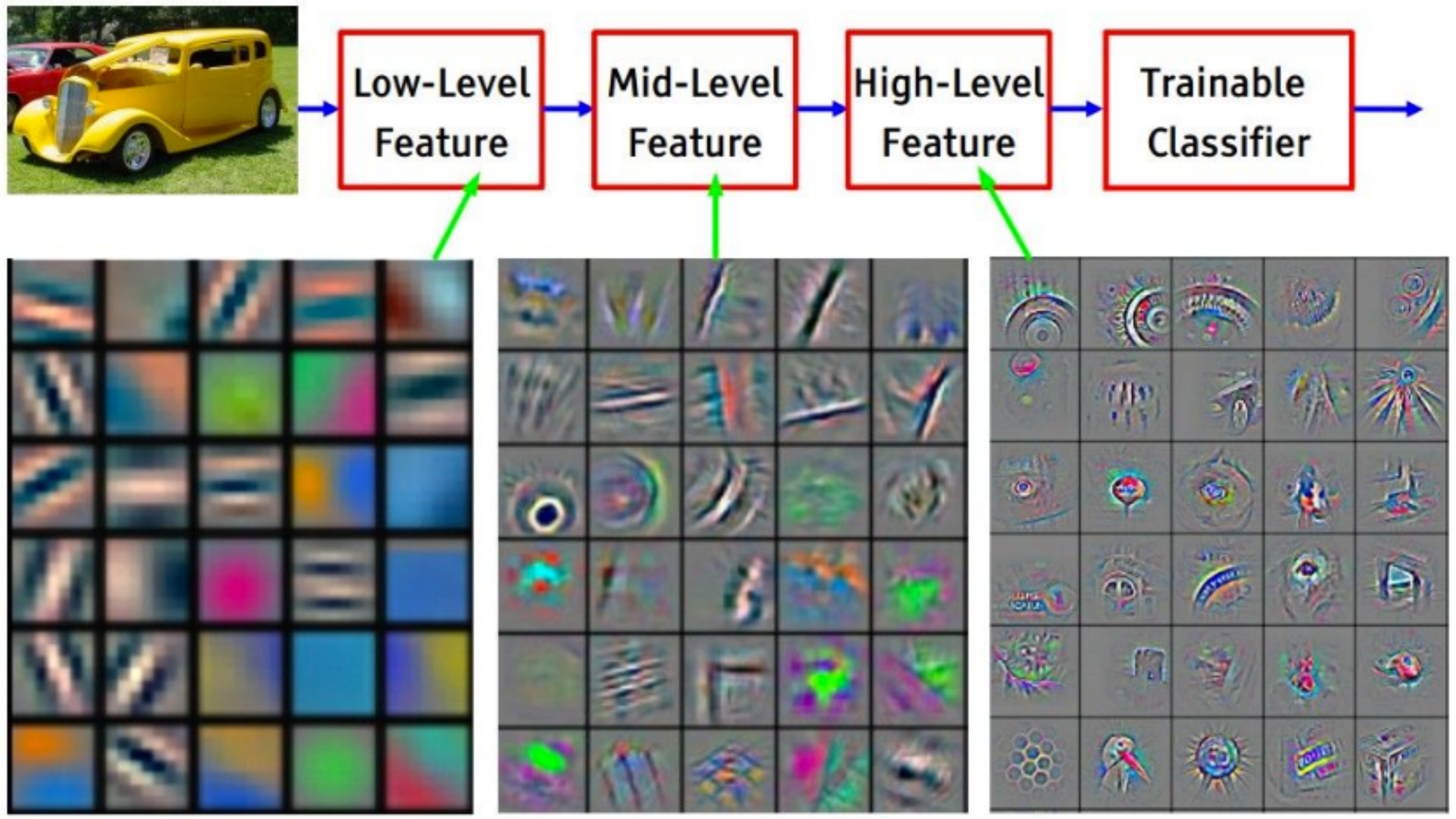
Convolutional Layers



RGB input

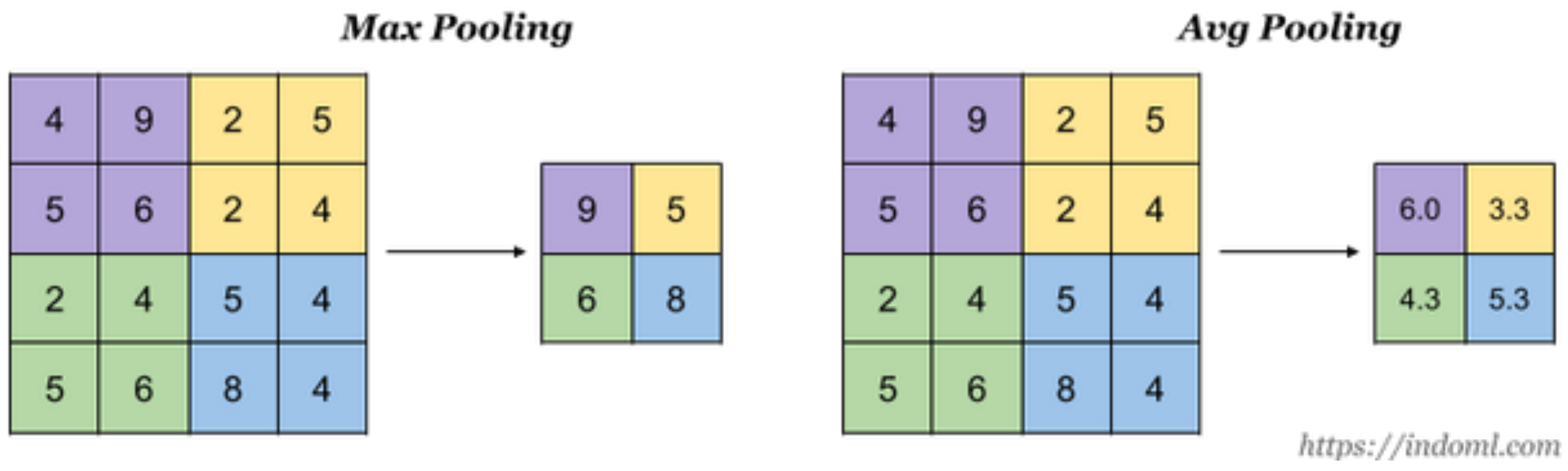
Convolutional layer output

Convolutional Filters

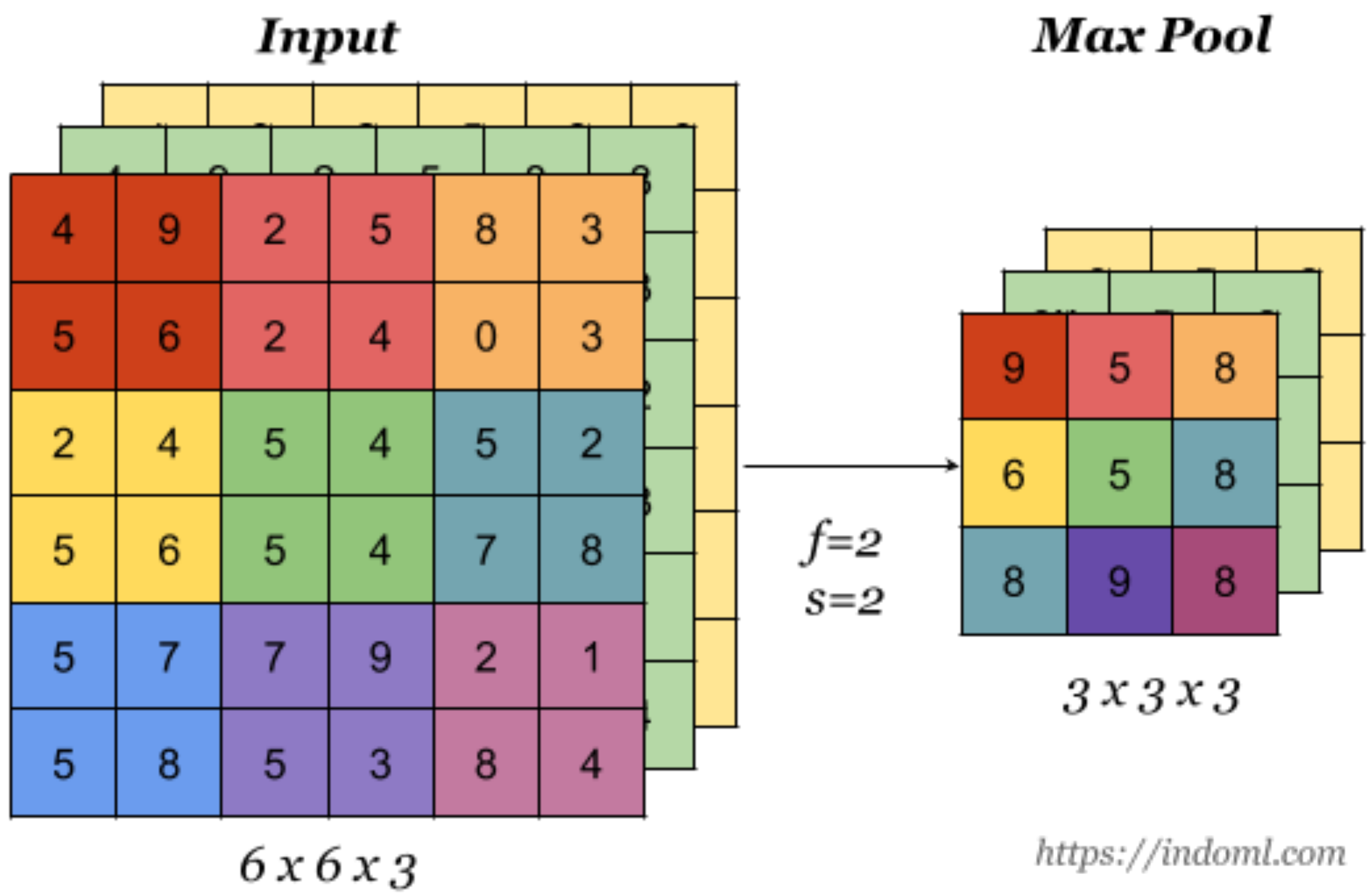


Pooling Layers

Pooling takes the maximum or average of a block of values. It is reduces the size of the hidden layers, speeds up calculations, and makes the features more robust.

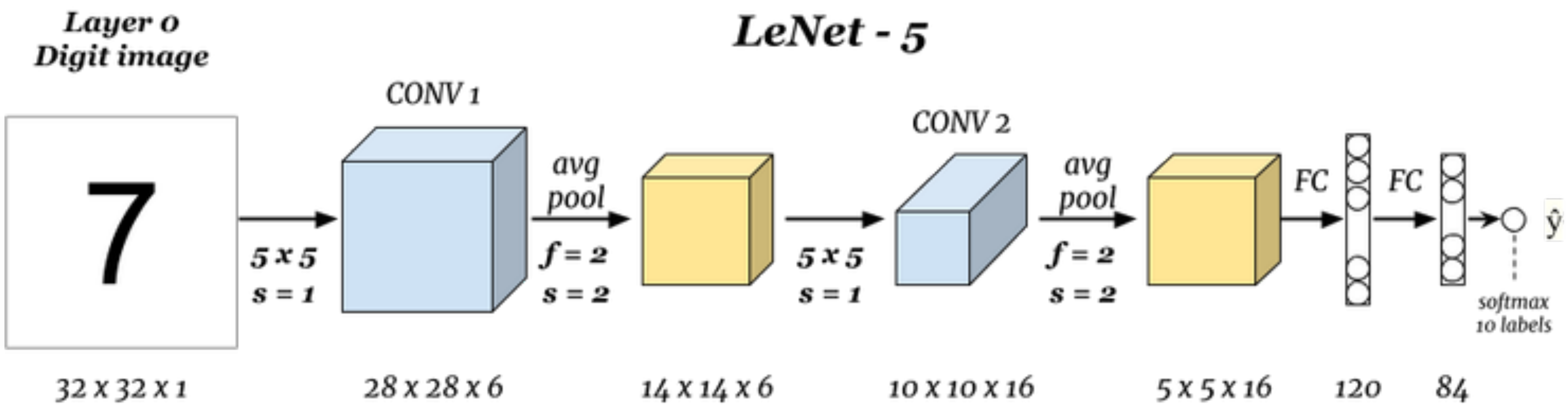


Pooling Layers

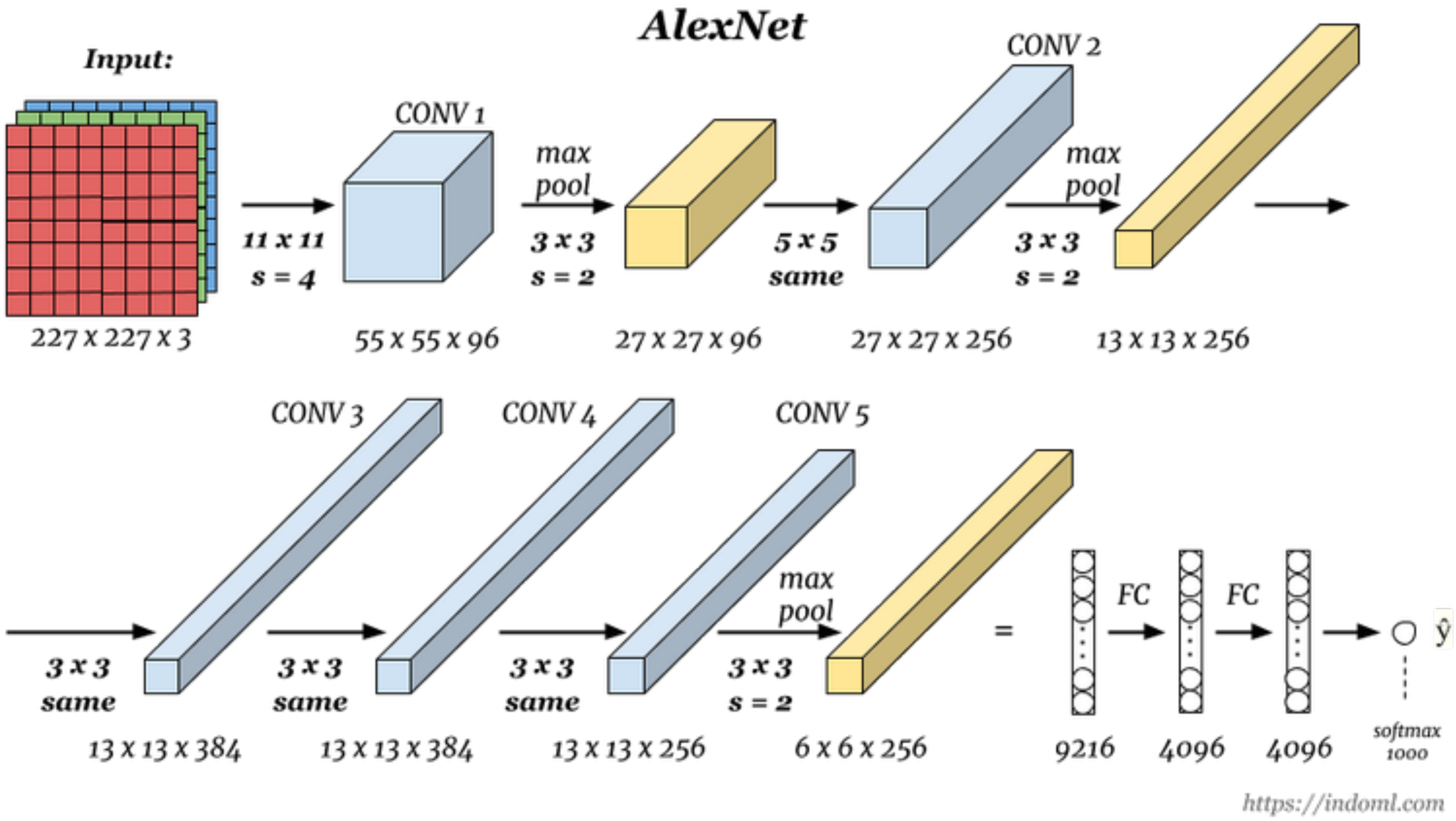


Max-pooling layer output

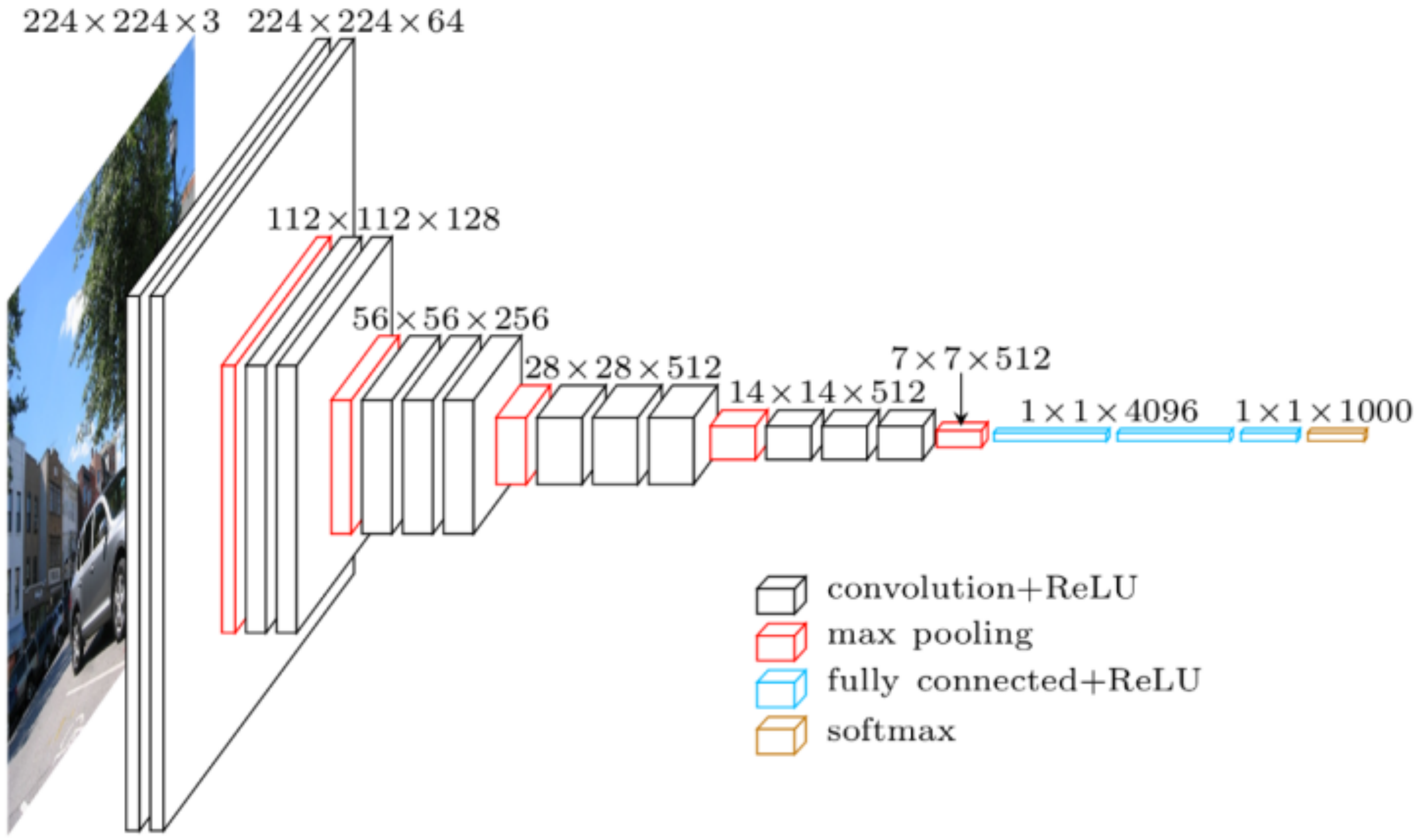
Example: LeNet-5 (1998)



Example: AlexNet (2012)

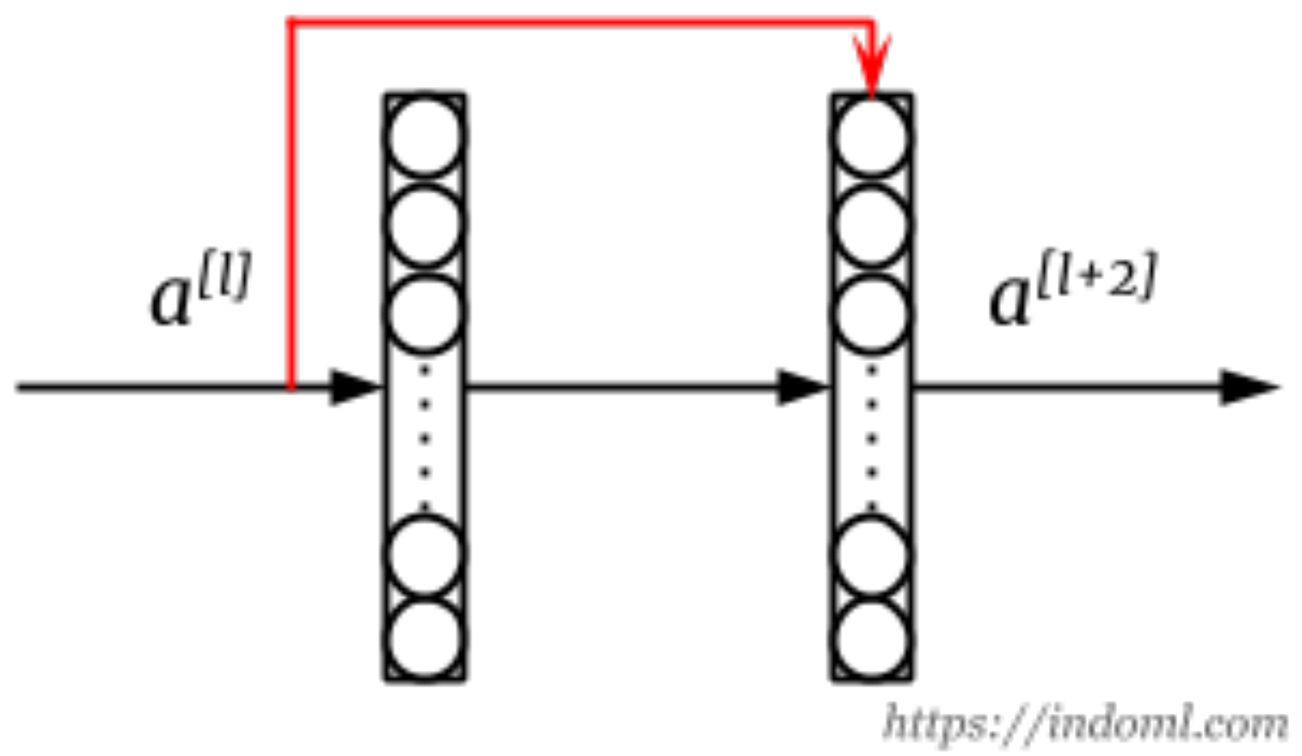


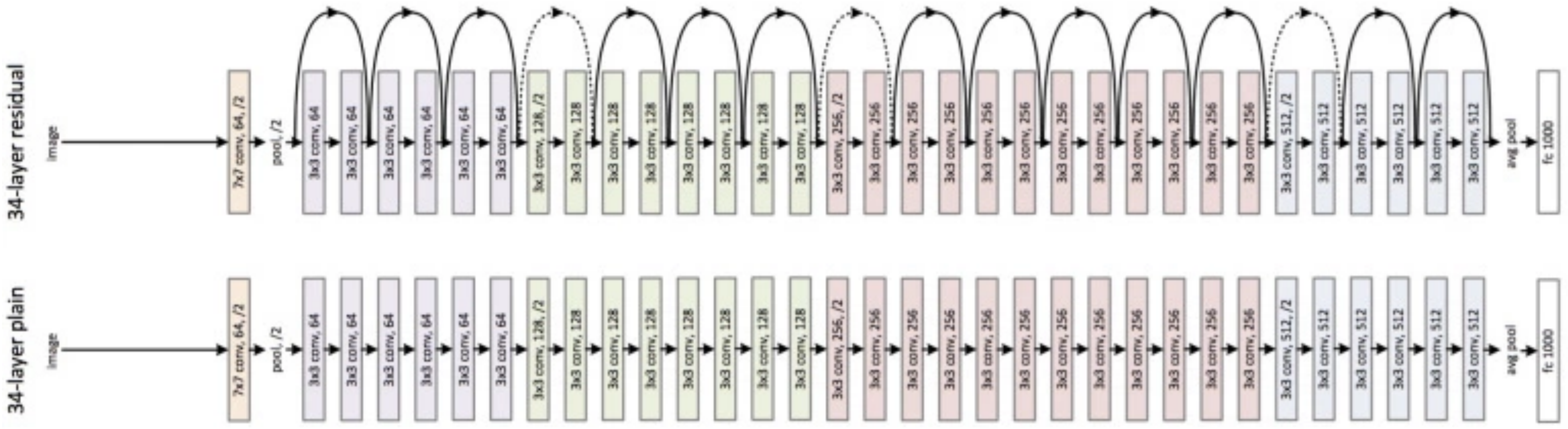
Example: VGG-16 (2014)



Example: ResNet (2015)

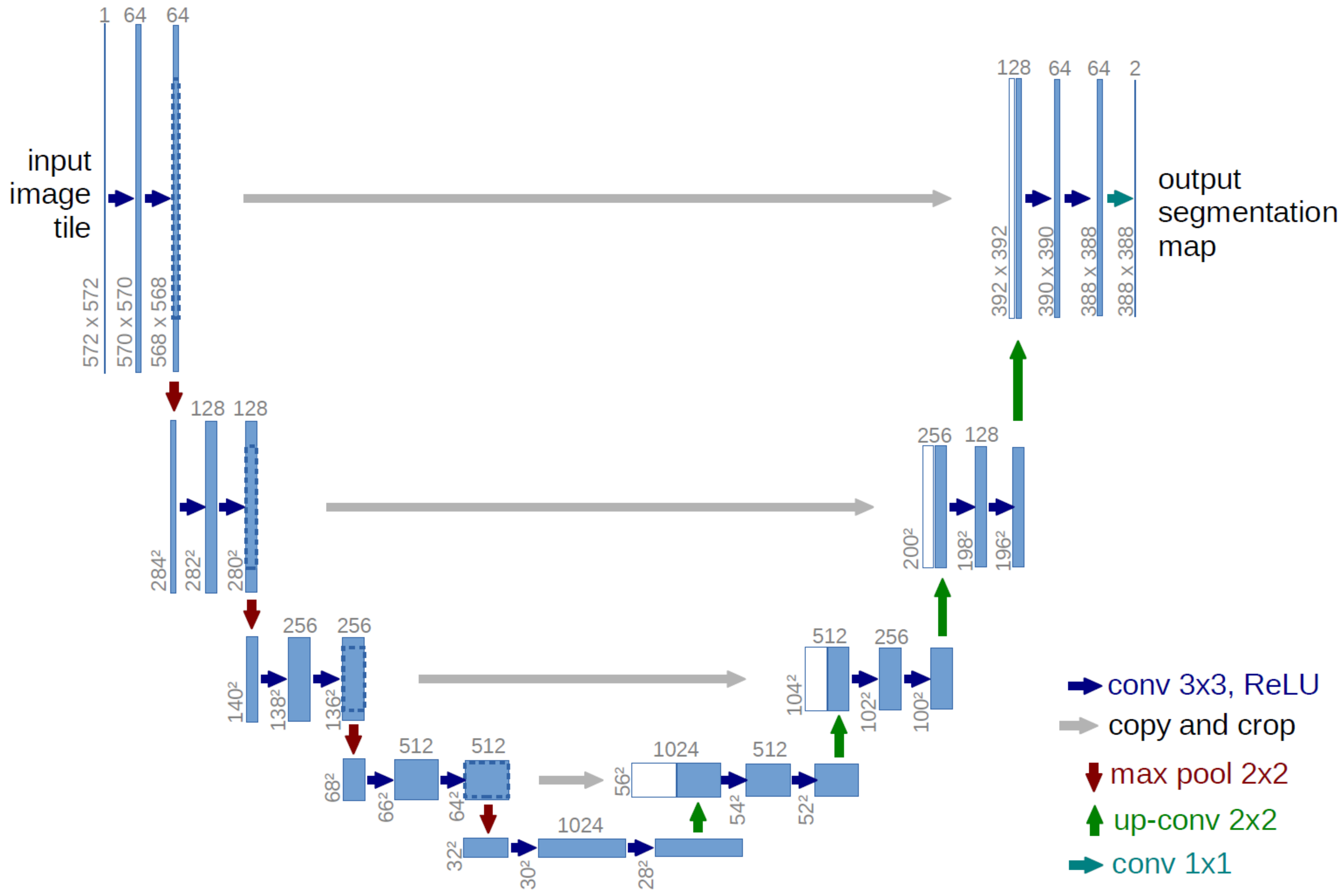
Deeper networks become harder to train. ResNet adds “skip connections” where output from one layer is fed to layer deeper in the network.





Example: U-Net (2015)

Max pooling layers downsample image resolution. To perform segmentation, upsample back to original resolution.



Recurrent Neural Networks (RNN)

Sequential Data as Inputs

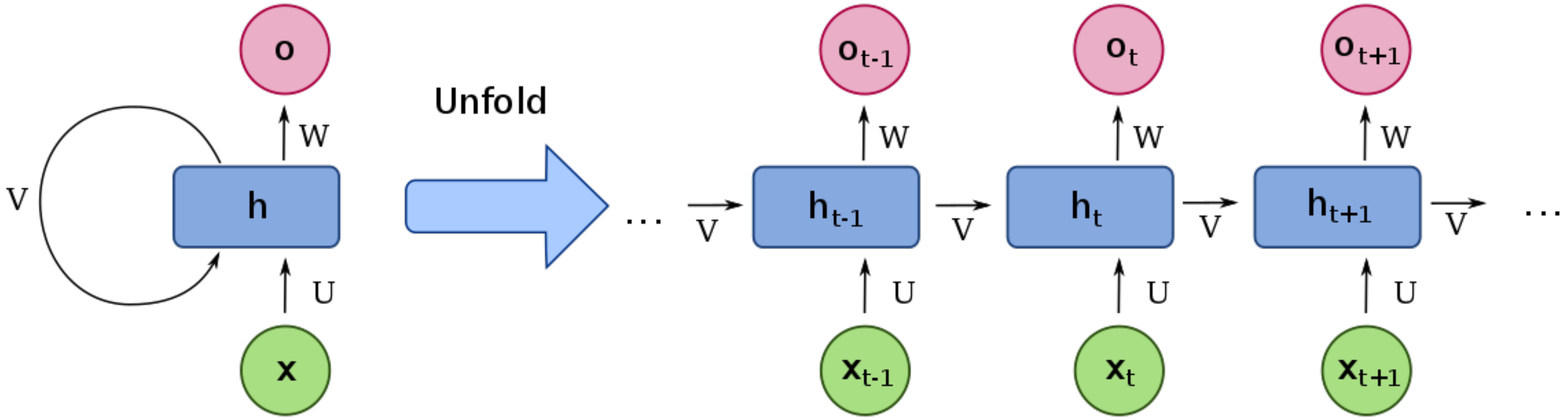
A sequence is a stream of data (finite or infinite, fixed or variable length) that are interdependent.

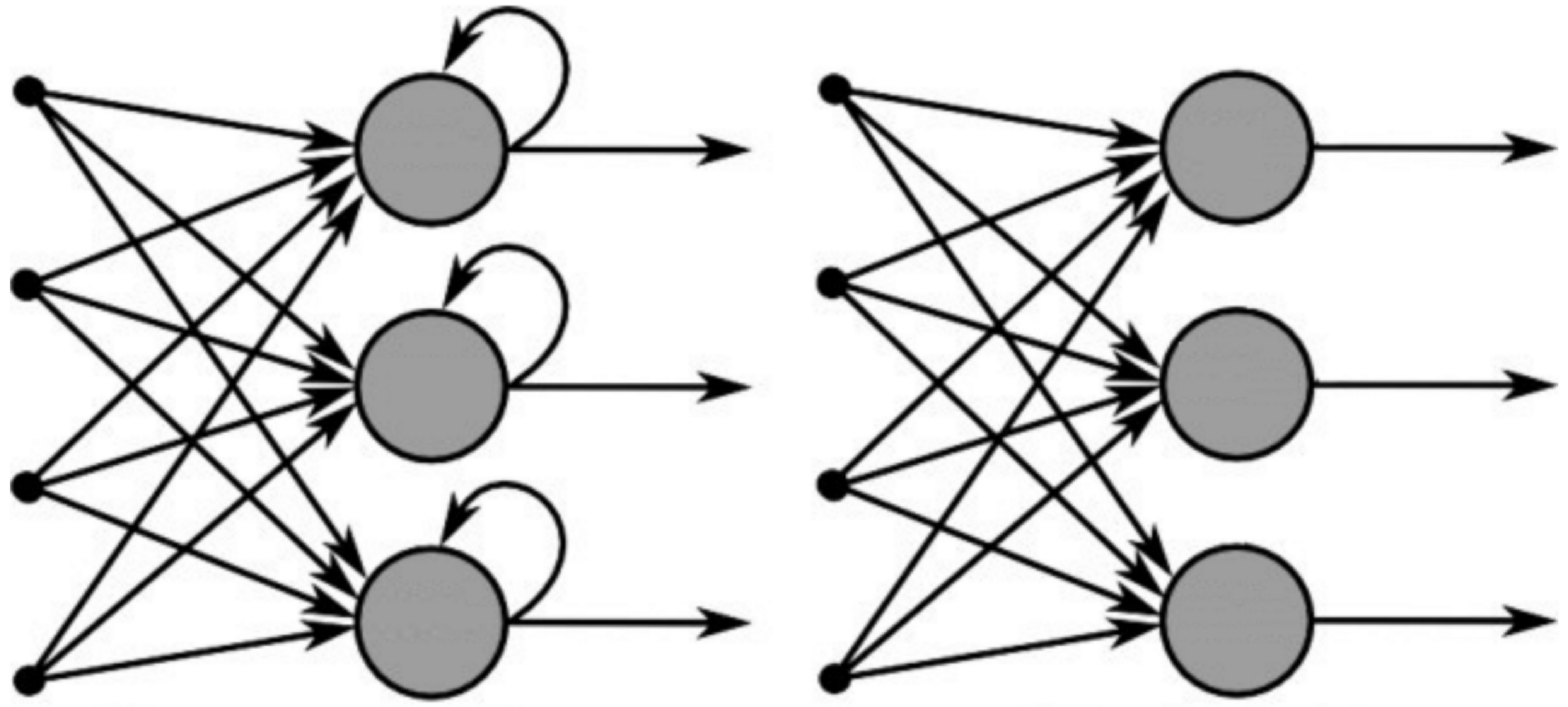
Examples include text, speech, any time series data.

We want a network that “remembers” what it has seen so far when processing the next item of the input.

Recurrent Neural Networks

At each time step t, the RNN takes as input the raw input at t and the output of the hidden layers at time t-1. Not a feedforward network — their own outputs become inputs again at the next time step.

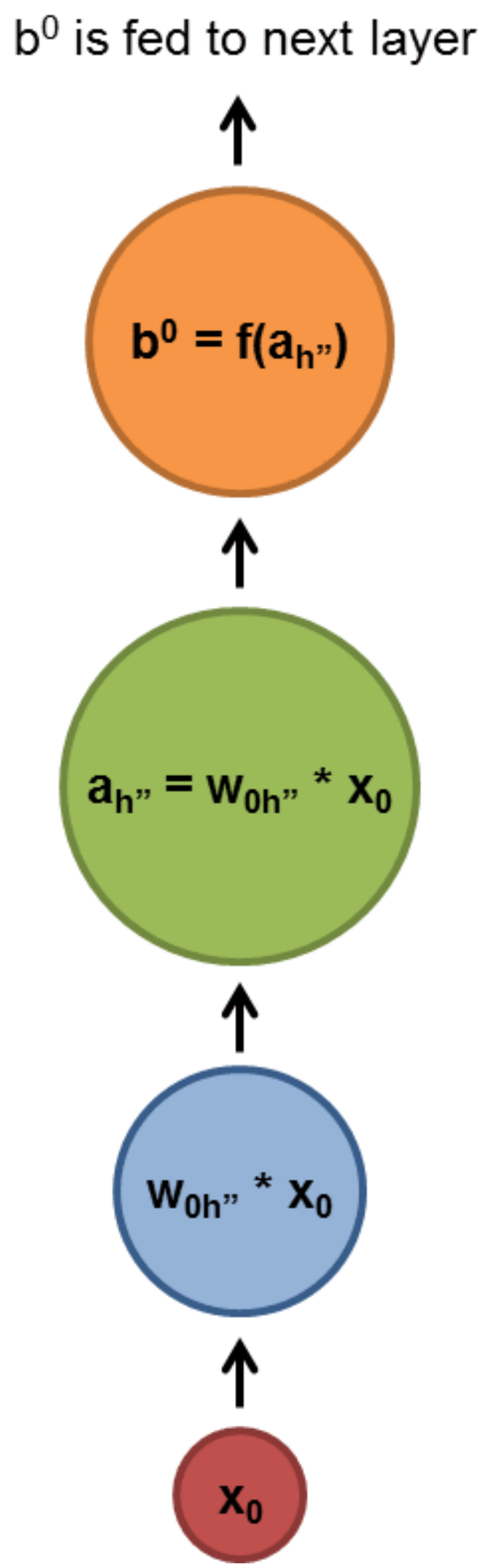




Recurrent Neural Network   Feed-Forward Neural Network

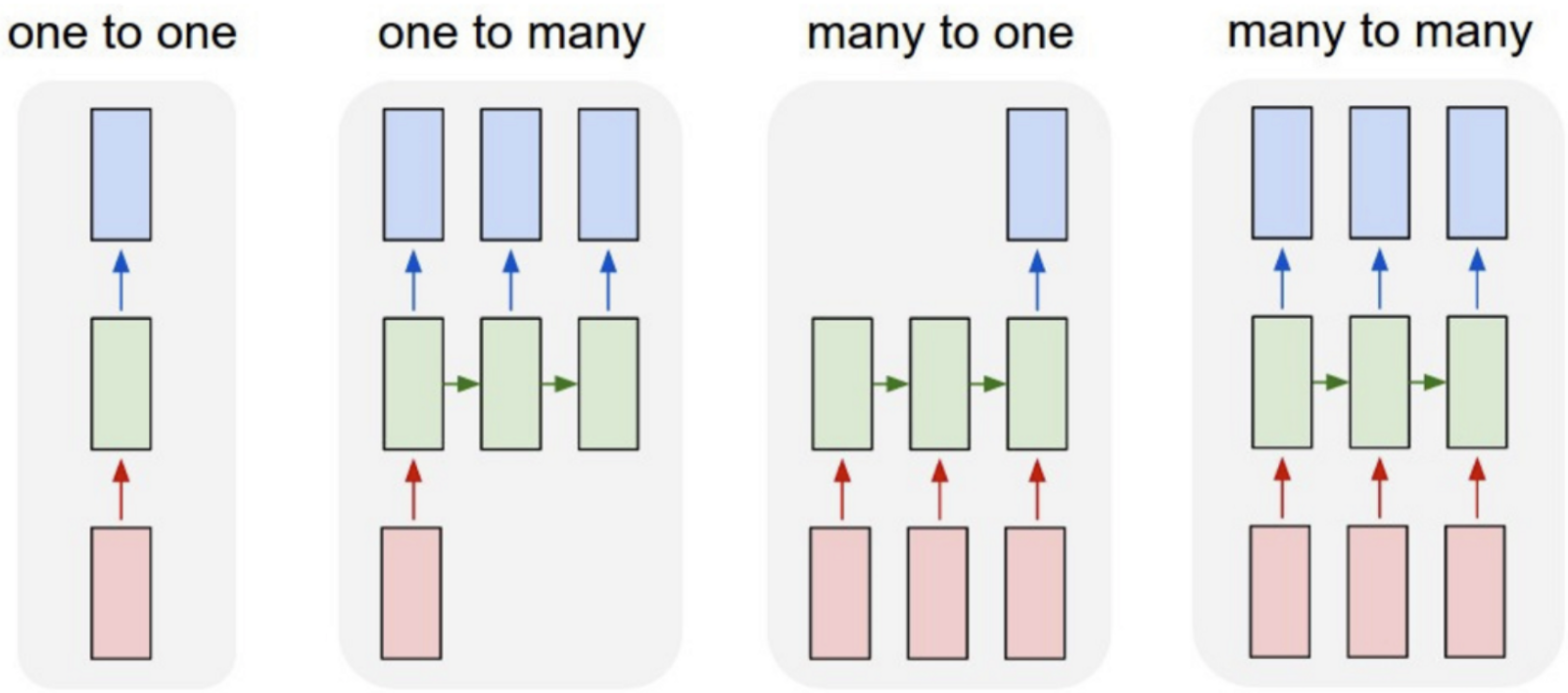
https://towardsdatascience.com/recurrent-neural-networks-and-lstm-4b601dd822a5

Recurrent Neural Networks



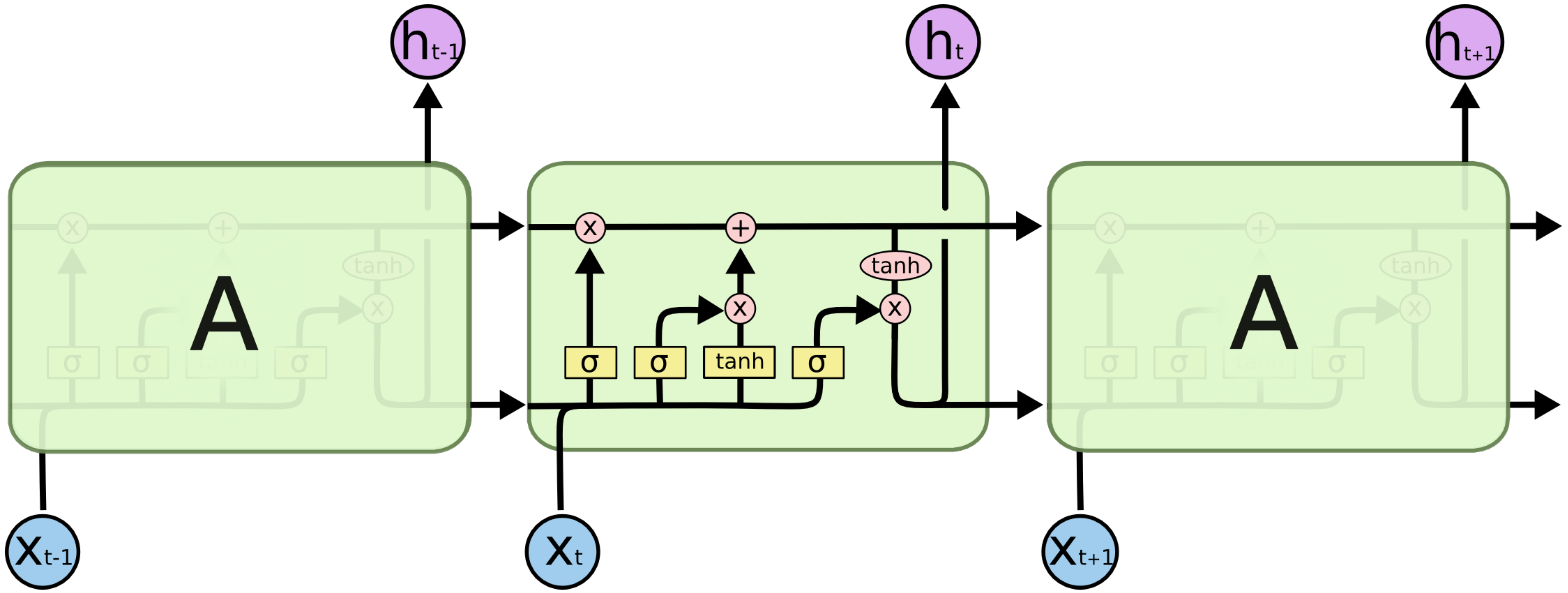
Recurrent Neural Networks

Where a feedforward NN maps one input to one output, RNNs can map one to many, many to one, or many to many.



Long-Short Term Memory (LSTM)

An architecture that enables RNNs to remember inputs over a long period of time.



http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Implementing Neural Networks

Available Software Libraries

Python:

- `sklearn` Multi-layer perceptron
- TensorFlow
- Keras
- PyTorch

R: neuralnet package, Keras

Matlab: Deep Learning Toolbox

Example Code

PyTorch

```python
class AlexNet(nn.Module):

    def __init__(self, num_classes=1000):
        super(AlexNet, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(64, 192, kernel_size=5, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(192, 384, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
        )
        self.avgpool = nn.AdaptiveAvgPool2d((6, 6))
        self.classifier = nn.Sequential(
            nn.Dropout(),
            nn.Linear(256 * 6 * 6, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, num_classes),
        )

    def forward(self, x):
        x = self.features(x)
        x = self.avgpool(x)
        x = x.view(x.size(0), 256 * 6 * 6)
        x = self.classifier(x)
        return x
```